Kwangkeun Yi (Ed.)

LNCS 3780

Programming Languages and Systems

Third Asian Symposium, APLAS 2005
Tsukuba, Japan, November 2005
Proceedings

Springer

Kwangkeun Yi (Ed.)

Programming Languages and Systems

Third Asian Symposium, APLAS 2005
Tsukuba, Japan, November 2-5, 2005
Proceedings

Springer

Volume Editor

Kwangkeun Yi
Seoul National University
School of Computer Science and Engineering, Seoul 151-742, Korea
E-mail: kwang@cse.snu.ac.kr

Library of Congress Control Number: 2005934457

CR Subject Classification (1998): D.3, D.2, F.3, D.4, D.1, F.4.1

ISSN	0302-9743
ISBN-10	3-540-29735-9 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-29735-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11575467 06/3142 5 4 3 2 1 0

Foreword

APLAS 2005 was the Asian Symposium on Programming Languages and Systems, held in Tsukuba, Japan in November 2–5, 2005. It was the latest event in the series of annual meetings started in 2000 by Asian researchers in the field of programming languages and systems. The first three were organized as workshops, and were held in Singapore (2000), Daejeon (2001), and Shanghai (2002). The enthusiasm there, encouraged by the rich production of original research papers, and the support of the international research communities in programming languages and systems, led to the first APLAS as a symposium in Beijing (2003), followed by the one in Taipei (2004). APLAS 2005 was the third symposium in the series.

In the past five years we have witnessed the growing role of APLAS as one of the key research communities of the world. This is not only because Asia and the Pacific Rim is a fast-growing region of IT industries, but because it is a region of highly cultivated human resources. We are confident that forums like APLAS will further engender interaction among Asian researchers and with the rest of the world.

As for the scope of APLAS, from the very beginning we have been striving to achieve the cross-fertilization of theories and system developments of programming and programming languages. The papers selected for the publication of this volume of the proceedings are the evidence of our efforts. We are very grateful to the contributors of the submitted papers; they came not only from Asia and Australia but from Europe and North America.

The symposium was held in the campus of the University of Tsukuba. Bringing APLAS 2005 to its realization was the result of the collaborative work of the Organizing Committee and various organizations that supported us. Foremost, I would like to express my thanks for the work of the Program Committee chaired by Kwangkeun Yi, who was responsible for the collection of high-quality papers on the state of the art of the research in programming languages and systems. We had three invited talks, by Patrick Cousot, Haruo Hosoya and Thomas Reps, to whom we are very grateful. We would like to express our thanks for the support by AAFS (Asian Association for Foundation of Software), JSSST (Japan Society for Software Science and Technology), IISF (International Information Science Foundation), and the University of Tsukuba. Last but not least, I would like to express our thanks to Springer for its continued support in the publication of the proceedings in the Lecture Notes in Computer Science series.

Septermber 2005 Tetsuo Ida

Preface

This volume contains the proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS 2005), which took place in Tsukuba, Japan, November 2–5, 2005. The symposium was sponsored by the Asian Association for Foundation of Software (AAAF), Japan Society for Software Science and Technology (JSSST), International Information Science Foundation in Japan (IISF), and University of Tsukuba.

Among the 78 full submissions, the Program Committee selected 24 papers. Almost all submissions were reviewed by three (or more) PC members with the help of external reviewers. Papers were selected during a 16-day electronic discussion phase. I would like to sincerely thank the members of the APLAS 2005 Program Committee for their thorough reviews and dedicated involvement during the discussion phase. I would also like to thank all the external reviewers for their invaluable contributions.

The symposium program covered both theoretical and practical topics in programming languages and systems. In addition to the 24 accepted papers, the symposium also featured invited talks by three distinguished speakers: Patrick Cousot (École Normale Superieure, Paris, France), Haruo Hosoya (University of Tokyo, Japan), and Thomas Reps (University of Wisconsin, Madison, USA).

Many people helped to establish APLAS as a high-quality forum in Asia that serves programming languages and systems researchers worldwide. The APLAS series started after informal yet well-attended workshops in Singapore (2000), Daejeon (2001), and Shanghai (2002). After the workshops, the first two formal symposiums were held in Beijing (2003) and Taipei (2004). This third symposium inherited much from this past momentum and owed many people for its success.

I am grateful to the General Chair, Tetsuo Ida, for his invaluable support and guidance that made our symposium in Tsukuba possible and enjoyable. I am indebted to our Local Arrangements Chair, Mircea Marin, for his considerable effort to plan and organize the meeting itself. I thank Hongseok Yang for serving as the Poster Chair. Last but not least, I also thank Deokhwan Kim for his help in handling the OpenConf system and preparing these proceedings.

September 2005 Kwangkeun Yi

Organization

General Chair

Tetsuo Ida (University of Tsukuba, Japan)

Program Chair

Kwangkeun Yi (Seoul National University, Korea)

Program Committee

Radhia Cousot (CNRS/École Polytechnique, France)
Manuel Fahndrich (Microsoft Research, USA)
Masami Hagiya (Univ. of Tokyo, Japan)
Luddy Harrison (Univ. of Illinois at Urbana-Champaign, USA)
Siau-Cheng Khoo (Nat. Univ. of Singapore, Singapore)
Naoki Kobayashi (Tohoku Univ., Japan)
Oukseh Lee (Hanyang Univ., Korea)
Peter Lee (Carnegie Mellon Univ., USA)
Huimin Lin (Chinese Academy of Sciences, China)
Soo-mook Moon (Seoul National Univ., Korea)
Alan Mycroft (Univ. of Cambridge, UK)
Atsushi Ohori (Tohoku Univ., Japan)
David Schmidt (Kansas State Univ., USA)
Harald Søndergaard (Univ. of Melbourne, Australia)
Martin Sulzmann (National Univ. of Singapore, Singapore)
Wuu Yang (National Chiao-Tung Univ., Taiwan)
Wang Yi (Uppsala Univ., Sweden)

Local Arrangements Chair

Mircea Marin (University of Tsukuba, Japan)

Poster Chair

Hongseok Yang (Seoul National University, Korea)

External Referees

Tatsuya Abe
Shivali Agarwal
Hugh Anderson
Stefan Andrei
Krzysztof Apt
David Aspinall
Michael Baldamus
Massimo Bartoletti
Manuel Chakravarty
Haiming Chen
Kung Chen
Wei-Ngan Chin
Hyung-Kyu Choi
Kwanghoon Choi
Amy Corman
Patrick Cousot
Alexandre David
Stephen A. Edwards
Robert Ennals
Erik Ernst
Jerome Feret
Carl Frederiksen
Alain Frisch
Craciun M.F. Gabriel
Marisa Gil
Sebastian Hack
Martin Henz
Ben Horsfall
Pao-Ann Hsiung
John Håkansson
Atsushi Igarashi
Sasano Isao
Stanislaw Jarzabek
Neil Johnson
Andrew Kennedy
Suhyun Kim
Nils Klarlund
Pavel Krcal
Julia Lawall
Alan Lawrence
Je-Hyung Lee
Junpyo Lee
Seung-Il Lee
Yongjian Li
Xinxin Liu
Francesco Logozzo
Anton Lokhmotov
Kenny Z.M. Lu
Stephen Magill
Damien Massé
Isabella Mastroeni
Yutaka Matsuno
Leonid Mokrushin
David Monniaux
Shin-Cheng Mu
Aleks Nanevsky
Nicholas Nethercote
Luke Ong
YunHeung Paek
Paul Pettersson
Corneliu Popeea
Shengchao Qin
Julian Rathke
Davide Sangiorgi
Peter Schachte
Peter Sestoft
Zhong Shao
Jeremy Singer
Rafal Somla
Eijiro Sumii
Koichi Takahashi
Eben Upton
Viktor Vafeiadis
Arnaud Venet
Razvan Voicu
David Walker
Meng Wang
Andrzej Wasowski
Jeremy Wazny
Weng-Fai Wong
Dana N. Xu
Mitsuharu Yamamoto
Roland Yap
Steve Zdancewic
Naijun Zhan
Wenhui Zhang

Sponsoring Institutions

Asian Association for Foundation of Software (AAFS)
Japan Society for Software Science and Technology (JSSST)
International Information Science Foundation, Japan (IISF)
University of Tsukuba

Table of Contents

Type Systems for XML

Haruo Hosoya

The University of Tokyo

Abstract. XML is a standard data format that is nowadays used everywhere. A notable feature of XML is its user-definable schemas. Schemas describe structural constraints on XML documents, thus defining “types” of XML. However, in current languages and systems for processing XML, those types are used only for dynamically validating data, not for statically verifying programs.

The goal of this work is to establish methods for the design and implementation of type systems for XML processing. However, this task is not a simple transfer of existing knowledges in programming languages since XML types are based on regular tree expressions and therefore have much richer structure than standard types treated in past researches. More concretely, difficulties arise in finding suitable definitions and algorithms for (1) typing concepts already standard in functional programming, e.g., subtyping and parametric polymorphism, (2) XML-specific structures, e.g., (in addition to regular tree expressions) attribute and shuffle expressions, and (3) language constructs for XML processing, e.g., pattern matching and its extensions. In this talk, I will overview our efforts dealing with these issues, emphasizing the principles consistently used in all of these—”definition by semantics” and “implementation based on finite tree automata.”

This work has been done jointly with Jérôme Vouillon, Benjamin Pierce, Makoto Murata, Tadahiro Suda, Alain Frisch, and Giuseppe Castagna.

The Essence of Dataflow Programming
(Short Version)

Tarmo Uustalu[1] and Varmo Vene[2]

[1] Inst. of Cybernetics at Tallinn Univ. of Technology,
Akadeemia tee 21, EE-12618 Tallinn, Estonia
tarmo@cs.ioc.ee

[2] Dept. of Computer Science, Univ. of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia
varmo@cs.ut.ee

Abstract. We propose a novel, comonadic approach to dataflow (stream-based) computation. This is based on the observation that both general and causal stream functions can be characterized as coKleisli arrows of comonads and on the intuition that comonads in general must be a good means to structure context-dependent computation. In particular, we develop a generic comonadic interpreter of languages for context-dependent computation and instantiate it for stream-based computation. We also discuss distributive laws of a comonad over a monad as a means to structure combinations of effectful and context-dependent computation. We apply the latter to analyse clocked dataflow (partial stream based) computation.

1 Introduction

Ever since the work by Moggi and Wadler [22,35], we know how to reduce impure computations with errors and non-determinism to purely functional computations in a structured fashion using the maybe and list *monads*. We also know how to explain other types of *effect*, such as *continuations*, *state*, even *input/output*, using monads!

But what is more unnatural or hard about the following program?

```
pos  = 0 fby (pos + 1)
fact = 1 fby (fact * (pos + 1))
```

This represents a *dataflow* computation which produces two discrete-time signals or streams: the enumeration of the naturals and the graph of the factorial function. The syntax is essentially that of Lucid [2], which is an old *intensional* language, or Lustre [15] or Lucid Synchrone [10,27], the newer *French synchronous dataflow* languages. The operator fby reads 'followed by' and means initialized unit delay of a discrete-time signal (cons of a stream).

Notions of dataflow computation cannot be structured by monads. As a substitute, one can use the laxer framework of *Freyd categories* or *arrow types*, proposed independently by Power and Robinson [28] and Hughes [17]. The message of this paper is that while this works, one can alternatively use something much more simple and standard, namely *comonads*, the formal dual of comonads. Moreover, comonads are even better, as there is more relevant structure to them than to Freyd categories. We also mean to claim that, compared to monads, comonads have received too little attention in programming language semantics. This is unfair, since just as monads are useful for speaking and reasoning about notions of functions that produce effects, comonads can handle context-dependent functions and are hence highly relevant. This has been suggested earlier, e.g., by Brookes and Geva [8] and Kieburtz [19], but never caught on because of a lack of compelling examples. But now dataflow computation provides clear examples and it hints at a direction in which there are more.

Technically, we show that general and causal stream functions, the basic entities in intensional and synchronous dataflow computation, are elegantly described in terms of comonads. Imitating monadic interpretation, we develop a generic comonadic interpreter for context-dependent computation. By instantiation, we obtain interpreters of a Lucid-like intensional language and a Lucid Synchrone-like synchronous dataflow language. Remarkably, we get higher-order language designs with almost no effort whereas the traditional dataflow languages are first-order and the question of the meaningfulness or right meaning of higher-order dataflow has been seen as controversial. We also show that clocked dataflow (i.e., partial-stream based) computation can be handled by distributive laws of the comonads for stream functions over the maybe monad.

The organization of the paper is as follows. In Section 2, we introduce comonads and argue that they structure computation with context-dependent functions. We show that both general and causal stream functions are smoothly described by comonads and develop a comonadic interpreter capable of handling dataflow languages. In Section 3, we show how effects and context-dependence can be combined in the presence of a distributive law of the comonad over the monad, show how this applies to partial-stream functions and present a distributivity-based interpreter which copes with clocked dataflow languages. Section 4 is a summary of related work, while Section 5 lists our conclusions.

We assume that the reader is familiar with the basics of functional programming (in particular, Haskell programming), denotational semantics and the Lambek-Lawvere correspondence between typed lambda calculi and cartesian closed categories (the types-as-objects, terms-as-morphisms correspondence). Acquaintance with dataflow programming (Lucid or Lucid Synchrone) will be of additional help. Monads and arrow types are not introduced in this short version of the paper, but comonads and distributive laws are.

The paper is related to our earlier paper [33], which discussed the relevance of comonads for dataflow computation but did not treat comonad-based processing of dataflow languages.

2 Comonads

2.1 Comonads and Context-Dependent Functions

We start by defining what comonads are and explaining their intuitive relevance for notions of impure computation.

A *comonad* on a category $\mathcal{C}$ is given by a mapping $D : |\mathcal{C}| \to |\mathcal{C}|$ together with a $|\mathcal{C}|$-indexed family ε of maps $\varepsilon_A : DA \to A$ (*counit*), and an operation $-^\dagger$ taking every map $k : DA \to B$ in $\mathcal{C}$ to a map $k^\dagger : DA \to DB$ (*coextension operation*) such that

1. for any $k : DA \to B$, $\varepsilon_B \circ k^\dagger = k$,
2. $\varepsilon_A{}^\dagger = \mathsf{id}_{DA}$,
3. for any $k : DA \to B$, $\ell : DB \to C$, $(\ell \circ k^\dagger)^\dagger = \ell^\dagger \circ k^\dagger$.

Analogously to Kleisli categories, any comonad $(D, \varepsilon, -^\dagger)$ defines a category $\mathcal{C}_D$ where $|\mathcal{C}_D| = |\mathcal{C}|$ and $\mathcal{C}_D(A, B) = \mathcal{C}(DA, B)$, $(\mathsf{id}_D)_A = \varepsilon_A$, $\ell \circ_D k = \ell \circ k^\dagger$ (*coKleisli category*) and an identity on objects functor $J : \mathcal{C} \to \mathcal{C}_D$ where $Jf = f \circ \varepsilon_A$ for $f : A \to B$.

Comonads should be fit to capture notions of "value in a context"; DA would be the type of contextually situated values of A. A context-dependent function from A to B would then be a map $A \to B$ in the coKleisli category, i.e., a map $DA \to B$ in the base category. The function $\varepsilon_A : DA \to A$ discards the context of its input whereas the coextension $k^\dagger : DA \to DB$ of a function $k : DA \to B$ essentially duplicates it (to feed it to k and still have a copy left).

Some examples of comonads are the following: each object mapping D below is a comonad:

– $DA = A$, the identity comonad,
– $DA = A \times E$, the product comonad,
– $DA = \mathsf{Str}A = \nu X.A \times X$, the streams comonad,
– $DA = \nu X.A \times FX$, the cofree comonad over F,
– $DA = \mu X.A \times FX$, the cofree recursive comonad over F [32].

Accidentally, the pragmatics of the product comonad is the same as that of the exponent monad, viz. representation of functions reading an environment. The reason is simple: the Kleisli arrows of the exponent monad are the maps $A \to (E \Rightarrow B)$ of the base category, which are of course in a natural bijection with the with the maps $A \times E \to B$ that are the coKleisli arrows of the product comonad. But in general, monads and comonads capture different notions of impure function. We defer the discussion of the pragmatics of the streams comonad until the next subsection (it is not the comonad to represent general or causal stream functions!).

For Haskell, there is no standard comonad library[1]. But of course comonads are easily defined as a type constructor class analogously to the definition of monads in Prelude and Monad.

[1] There is, however, a contributed library by Dave Menendez, see `http://www.eyrie.org/~zednenem/2004/hsce/`.

```
class Comonad d where
  counit :: d a -> a
  cobind :: (d a -> b) -> d a -> d b

cmap :: Comonad d => (a -> b) -> d a -> d b
cmap f = cobind (f . counit)
```

The identity and product comonads are defined as instances in the following fashion.

```
instance Comonad Id where
  counit (Id a) = a
  cobind k d = Id (k d)

data Prod e a = a :& e

instance Comonad (Prod e) where
  counit (a :& _) = a
  cobind k d@(_ :& e) = k d :& e

askP :: Prod e a -> e
askP (_ :& e) = e

localP :: (e -> e) -> Prod e a -> Prod e a
localP g (a :& e) = (a :& g e)
```

The stream comonad is implemented as follows.

```
data Stream a = a :< Stream a                      -- coinductive

instance Comonad Stream  where
  counit (a :< _) = a
  cobind k d@(_ :< as) = k d :< cobind k as

nextS :: Stream a -> Stream a
nextS (a :< as) = as
```

2.2 Comonads for General and Causal Stream Functions

The general pragmatics of comonads introduced, we are now ready to discuss the representation of general and causal stream functions via comonads.

The first observation to make is that streams (discrete time signals) are naturally isomorphic to functions from natural numbers: $\mathsf{Str} A = \nu X.\, A \times X \cong (\mu X.\, 1 + X) \Rightarrow A = \mathsf{Nat} \Rightarrow A$. In Haskell, this isomorphism is implemented as follows:

```
str2fun :: Stream a -> Int -> a
str2fun (a :< as) 0 = a
str2fun (a :< as) (i + 1) = str2fun as i

fun2str :: (Int -> a) -> Stream a
fun2str f = fun2str' f 0

fun2str' f i = f i :< fun2str' f (i + 1)
```

General stream functions $\mathsf{Str} A \to \mathsf{Str} B$ are thus in natural bijection with maps $\mathsf{Nat} \Rightarrow A \to \mathsf{Nat} \Rightarrow B$, which, in turn, are in natural bijection with maps $(\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} \to B$, i.e., $\mathsf{FunArg}\,\mathsf{Nat}\,A \to B$ where $\mathsf{FunArg}\,S\,A = (S \Rightarrow A) \times S$. Hence, for general stream functions, a value from A in context is a stream (signal) over A together with a natural number identifying a distinguished stream position (the present time). Not surprisingly, the object mapping $\mathsf{FunArg}\,S$ is a comonad (in fact, it is the "state-in-context" comonad considered by Kieburtz [19]) and, what is of crucial importance, the coKleisli identities and composition as well as the coKleisli lifting of $\mathsf{FunArg}\,\mathsf{Nat}$ agree with the identities and composition of stream functions (which are really just function identities and composition) and with the lifting of functions to stream functions. In Haskell, the parameterized comonad FunArg and the interpretation of the coKleisli arrows of $\mathsf{FunArg}\,\mathsf{Nat}$ as stream functions are implemented as follows.

```
data FunArg s a = (s -> a) :# s

instance Comonad (FunArg s) where
  counit (f :# s) = f s
  cobind k (f :# s) = (\ s' -> k (f :# s')) :# s

runFA :: (FunArg Int a -> b) -> Stream a -> Stream b
runFA k as = runFA' k (str2fun as :# 0)

runFA' k d@(f :# i) = k d :< runFA' k (f :# (i + 1))
```

The comonad $\mathsf{FunArg}\,\mathsf{Nat}$ can also be presented equivalently without using natural numbers to deal with positions. The idea for this alternative presentation is simple: given a stream and a distinguished stream position, the position splits the stream up into a list, a value of the base type and a stream (corresponding to the past, present and future of the signal). Put mathematically, there is a natural isomorphism $(\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} \cong \mathsf{Str}\,A \times \mathsf{Nat} \cong (\mathsf{List}\,A \times A) \times \mathsf{Str}\,A$ where $\mathsf{List}\,A = \mu X.\ 1 + (A \times X)$ is the type of lists over a given type A. This gives us an equivalent comonad LVS for representing of stream functions with the following structure (we use snoc-lists instead of cons-lists to reflect the fact that the analysis order of the past of a signal will be the reverse direction of time):

```
data List a = Nil | List a :> a                   -- inductive

data LV  a = List a := a

data LVS a = LV a :| Stream a

instance Comonad LVS where
  counit (az := a :| as) = a
  cobind k d = cobindL d := k d :| cobindS d
     where cobindL (Nil        := a :| as)  = Nil
           cobindL (az' :> a'  := a :| as)  = cobindL d' :> k d'
                                    where d' = az' := a' :| (a :< as)
           cobindS (az := a :| (a' :< as')) = k d' :< cobindS d'
                                    where d' = az :> a := a' :| as'
```

(Notice the visual purpose of our constructor naming. In values of types $\mathsf{LVS}\,A$, both the cons constructors (:>) of the list (the past) and the cons constructors (:<) of the stream (the future) point to the present which is enclosed between the constructors (:=) and (:|).)

The interpretation of the coKleisli arrows of the comonad LVS as stream functions is implemented as follows.

```
runLVS :: (LVS a -> b) -> Stream a -> Stream b
runLVS k (a' :< as') = runLVS' k (Nil := a' :| as')

runLVS' k d@(az := a :| (a' :< as')) = k d :< runLVS' k (az :> a := a' :| as')
```

Delay and anticipation can be formulated for both FunArg Nat and LVS.

```
fbyFA :: a -> (FunArg Int a -> a)
fbyFA a0 (f :# 0)       = a0
fbyFA _  (f :# (i + 1)) = f i

nextFA :: FunArg Int a -> a
nextFA (f :# i) = f (i + 1)
```

```
fbyLVS :: a -> (LVS a -> a)
fbyLVS a0 (Nil       := _ :| _) = a0
fbyLVS _  ((_ :> a') := _ :| _) = a'

nextLVS :: LVS a -> a
nextLVS (_ := _ :| (a :< _)) = a
```

Let us call a stream function *causal*, if the present of the output signal only depends on the past and present of the input signal and not on its future[2].

Is there a way to ban non-causal functions? Yes, the comonad LVS is easy to modify so that exactly those stream functions can be represented that are causal. All that needs to be done is to remove from the comonad LVS the factor of the future. We are left with the object mapping LV where $\mathsf{LV}\,A = \mathsf{List}\,A \times A = (\mu X.\, 1 + A \times X) \times A \cong \mu X.\, A \times (1 + X)$, i.e., a non-empty list type constructor. This is a comonad as well and again the counit and the coextension operation are just correct in the sense that they deliver the desirable coKleisli identities, composition and lifting. In fact, the comonad LV is the cofree recursive comonad of the functor Maybe (we refrain from giving the definition of a recursive comonad here, this can be found in [32]). It may be useful to notice that the type constructor LV carries a monad structure too, but the Kleisli arrows of that monad have nothing to do with causal stream functions!

In Haskell, the non-empty list comonad LV is defined as follows.

```
instance Comonad LV where
  counit (_ := a) = a
  cobind k d@(az := _) = cobindL k az := k d
     where cobindL k Nil = Nil
           cobindL k (az :> a) = cobindL k az :> k (az := a)

runLV  :: (LV a -> b) -> Stream a -> Stream b
runLV  k (a' :< as') = runLV' k (Nil := a' :| as')

runLV' k (d@(az := a) :| (a' :< as')) = k d :< runLV' k (az :> a := a' :| as')
```

With the LV comonad, anticipation is no longer possible, but delay is unproblematic.

```
fbyLV :: a -> (LV a -> a)
fbyLV a0 (Nil       := _) = a0
fbyLV _  ((_ :> a') := _) = a'
```

Analogously to causal stream functions, one might also consider *anticausal* stream functions, i.e., functions for which the present value of the output signal only depends on the present and future values of the input signal. As

[2] The standard terminology is '*synchronous* stream functions', but we want to avoid it because 'synchrony' also refers to all signals being on the same clock and to the hypothesis of instantaneous reactions.

$A \times \mathsf{Str}\, A \cong \mathsf{Str}\, A$, it is not surprising now anymore that the comonad for anticausal stream functions is the comonad Str, which we introduced earlier and which is very canonical by being the cofree comonad generated by the identity functor. However, in real life, causality is much more relevant than anticausality!

2.3 Comonadic Semantics

Is the comonadic approach to context-dependent computation of any use? We will now demonstrate that it is indeed by developing a generic comonadic interpreter instantiable to various specific comonads, in particular to those that characterize general and causal stream functions. In the development, we mimic the monadic interpreter of Moggi and Wadler [22,35].

As the first thing we must fix the syntax of our object language. We will support a purely functional core and additions corresponding to various notions of context.

```
type Var = String

data Tm = V Var | L Var Tm | Tm :@ Tm | Rec Tm
        | N Integer | Tm :+ Tm | ... | Tm :== Tm | ... | TT | FF | ... | If Tm Tm Tm
        | Tm `Fby` Tm       -- specific for both general and causal stream functions
        | Next Tm           -- specific for general stream functions only
```

The type-unaware semantic domain contains integers, booleans and functions, but functions are context-dependent (coKleisli functions). Environments are lists of variable-value pairs as usual.

```
data Val d = I Integer | B Bool | F (d (Val d) -> Val d)

type Env d = [(Var, Val d)]
```

We will manipulate environment-like entities via the following functions[3].

```
empty :: [(a, b)]                              update :: a -> b -> [(a, b)] -> [(a, b)]
empty = []                                     update a b abs = (a, b) : abs

unsafeLookup :: Eq a => a -> [(a, b)] -> b
unsafeLookup a0 ((a, b) : abs) = if a0 == a then b else unsafeLookup a0 abs
```

And we are at evaluation. Of course terms must denote coKleisli arrows, so the typing of evaluation is uncontroversial.

```
class Comonad d => ComonadEv d where
  ev :: Tm -> d (Env d) -> Val d
```

But an interesting issue arises with evaluation of closed terms. In the case of a pure or a monadically interpreted language, closed terms are supposed to be evaluated in the empty environment. Now they must be evaluated in the empty environment placed in a context! What does this mean? This is easy to understand on the example of stream functions. By the types, evaluation of an

[3] The safe lookup, that maybe returns a value, will be unnecessary, since we can type-check an object-language term before evaluating it. If this succeeds, we can be sure we will only be looking up variables in environments where they really occur.

expression returns a single value, not a stream. So the stream position of interest must be specified in the contextually situated environment that we provide. Very suitably, this is exactly the information that the empty environment in a context conveys. So we can define:

```
emptyL :: Int -> List [(a, b)]                emptyS :: Stream [(a, b)]
emptyL 0       = Nil                          emptyS = empty :< emptyS
emptyL (i + 1) = emptyL i :> empty

evClosedLVS :: Tm -> Int -> Val LVS
evClosedLVS e i = ev e (emptyL i := empty :| emptyS)

evClosedLV  :: Tm -> Int -> Val LV
evClosedLV  e i = ev e (emptyL i := empty)
```

Back to evaluation. For most of the core constructs, the types tell us what the defining clauses of their meanings must be—there is only one thing we can write and that is the right thing. In particular, everything is meaningfully predetermined about variables, application and recursion. E.g., for a variable, we must extract the environment from its context (e.g., history), and then do a lookup. For an application, we must evaluate the function wrt. the given contextually situated environment and then apply it. But since, according to the types, a function wants not just an isolated argument value, but a contextually situated one, the function has to be applied to the coextension of the denotation of the argument wrt. the given contextually situated environment.

```
_ev :: ComonadEv d => Tm -> d (Env d) -> Val d
_ev (V x)        denv = unsafeLookup x (counit denv)
_ev (e :@ e')    denv = case ev e denv of
                          F f -> f (cobind (ev e') denv)
_ev (Rec e)      denv = case ev e denv of
                          F f -> f (cobind (_ev (Rec e)) denv)
_ev (N n)        denv = I n
_ev (e0 :+ e1)   denv = case ev e0 denv of
                          I n0 -> case ev e1 denv of
                            I n1 -> I (n0 + n1)
...
_ev TT           denv = B True
_ev FF           denv = B False
_ev (If e e0 e1) denv = case ev e denv of B b -> if b then ev e0 denv else ev e1 denv
```

There is, however, a problem with lambda-abstraction. For any potential contextually situated value of the lambda-variable, the evaluation function should recursively evaluate the body of the lambda-abstraction expression in the appropriately extended contextually situated environment. Schematically,

```
_ev (L x e) denv = F (\ d -> ev e (extend x d denv))
```

where

```
extend :: Comonad d => Var -> d (Val d) -> d (Env d) -> d (Env d)
```

Note that we need to combine a contextually situated environment with a contextually situated value. One way to do this would be to use the strength of the comonad (we are in Haskell, so every comonad is strong), but in the case of the stream function comonads this would necessarily have the bad effect that either the history of the environment or that of the value would be lost. We would like to see that no information is lost, to have the histories zipped.

To solve the problem, we consider comonads equipped with an additional zipping operation. We define a *comonad with zipping* to be a comonad D coming with a natural transformation m with components $m_{A,B} : DA \times DB \to D(A \times B)$ that satisfies coherence conditions such as $\varepsilon_{A \times B} \circ m_{A,B} = \varepsilon_A \times \varepsilon_B$ (more mathematically, this is a symmetric semi-monoidal comonad).

In Haskell, we define a corresponding type constructor class.

```
class Comonad d => ComonadZip d where
  czip :: d a -> d b -> d (a, b)
```

The identity comonad, as well as LVS and LV are instances (and so are many other comonads).

```
instance ComonadZip Id  where
  czip (Id a) (Id b) = Id (a, b)

zipL :: List a -> List b -> List (a, b)
zipL Nil       _         = Nil
zipL _         Nil       = Nil
zipL (az :> a) (bz :> b) = zipL az bz :> (a, b)

zipS :: Stream a -> Stream b -> Stream (a, b)
zipS (a :< as) (b :< bs) = (a, b) :< zipS as bs

instance ComonadZip LVS where
  czip (az := a :| as) (bz := b :| bs) = zipL az bz := (a, b) :| zipS as bs

instance ComonadZip LV  where
  czip (az := a)       (bz := b)       = zipL az bz := (a, b)
```

With the zip operation available, defining the meaning of lambda-abstractions is easy, but we must also update the typing of the evaluation function, so that zippability becomes required.

```
class ComonadZip d => ComonadEv d where ...

_ev (L x e) denv = F (\ d -> ev e (cmap repair (czip d denv)))
                  where repair (a, env) = update x a env
```

It remains to define the meaning of the specific constructs of our example languages. The pure language has none. The dataflow languages have Fby and Next that are interpreted using the specific operations of the corresponding comonads. Since each of Fby and Next depends on the context of the value of its main argument, we need to apply the coextension operation to the denotation of that argument to have this context available.

```
instance ComonadEv Id  where
  ev e             denv = _ev e denv

instance ComonadEv LVS where
  ev (e0 `Fby` e1) denv = ev e0 denv `fbyLVS` cobind (ev e1) denv
  ev (Next e)      denv = nextLVS (cobind (ev e) denv)
  ev e             denv = _ev e denv

instance ComonadEv LV where
  ev (e0 `Fby` e1) denv = ev e0 denv `fbyLV`  cobind (ev e1) denv
  ev e denv = _ev e denv
```

In dataflow languages, the ‘followed by’ construct is usually defined to mean the delay of the second argument initialized by the initial value of the first argument, which may at first seem like an ad hoc decision (or so it seemed to us at least). Why give the initial position any priority? In our interpreter, we took the simplest possible solution of using the value of the first argument of Fby in the present position of the history of the environment. We did not use any explicit means to calculate the value of that argument wrt. the initial position. But the magic of the definition of fbyLVS is that it only ever uses its first argument when the second has a history with no past (which corresponds to the situation when the present actually is the initial position in the history of the environment). So our most straightforward naive design gave exactly the solution that has been adopted by the dataflow languages community, probably for entirely different reasons.

Notice also that we have obtained a generic comonads-inspired language design which supports higher-order functions and the solution was dictated by the types. This is remarkable since dataflow languages are traditionally first-order and the question of the right meaning of higher-order dataflow has been considered controversial. The key idea of our solution can be read off from the interpretation of application: the present value of a function application is the present value of the function applied to the history of the argument.

We can test the interpreter on a few classic examples from dataflow programming. The following examples make sense in both the general and causal stream function settings.

```
-- pos    = 0 fby pos + 1
pos  = Rec (L "pos" (N 0 ‘Fby‘ (V "pos" :+ N 1)))
-- sum x  = x + (0 fby sum x)
sum  = L "x" (Rec (L "sumx" (V "x" :+ (N 0 ‘Fby‘ V "sumx"))))
-- diff x = x - (0 fby x)
diff = L "x" (V "x" :- (N 0 ‘Fby‘ V "x"))
-- ini  x = x fby ini x
ini  = L "x" (Rec (L "inix" (V "x" ‘Fby‘ V "inix")))
-- fact = 1 fby (fact * (pos + 1))
fact = Rec (L "fact" (N 1 ‘Fby‘ (V "fact" :* (pos :+  N 1))))
-- fibo = 0 fby (fibo + (1 fby fibo))
fibo = Rec (L "fibo" (N 0 ‘Fby‘ (V "fibo" :+ (N 1 ‘Fby‘ V "fibo"))))
```

Testing gives expected results:

```
> runLV (ev pos) emptyS
0 :< (1 :< (2 :< (3 :< (4 :< (5 :< (6 :< (7 :< (8 :< (9 :< (10 :< ...
> runLV (ev (sum :@ pos)) emptyS
0 :< (1 :< (3 :< (6 :< (10 :< (15 :< (21 :< (28 :< (36 :< (45 :< (55 :< ...
> runLV (ev (diff :@ (sum :@ pos)) emptyS
0 :< (1 :< (2 :< (3 :< (4 :< (5 :< (6 :< (7 :< (8 :< (9 :< (10 :< ...
```

Here are two examples that are only allowed with general stream functions, because of using anticipation: the ‘whenever’ operation and the sieve of Eratosthenes.

```
-- x wvr y = if ini y then x fby (next x wvr next y) else (next x wvr next y)
wvr =  Rec (L "wvr" (L "x" (L "y" (
             If (ini :@ V "y")
               (V "x" ‘Fby‘ (V "wvr" :@ (Next (V "x")) :@ (Next (V "y"))))
```

```
                (V "wvr" :@ (Next (V "x")) :@ (Next (V "y")))))))
-- sieve x = x fby sieve (x wvr x mod (ini x) :/= N 0)
sieve = Rec (L "sieve" (L "x" (
              V "x" `Fby` (V "sieve" :@ (
                wvr :@ V "x" :@ (V "x" `Mod` (ini :@ (V "x")) :/= N 0))))))
-- eratosthenes = sieve (pos + 2)
eratosthenes = sieve :@ (pos :+ N 2)
```

Again, testing gives what one would like to get.

```
> runLVS (ev eratosthenes) emptyS
2 :< (3 :< (5 :< (7 :< (11 :< (13 :< (17 :< (19 :< (23 :< (29 :< ...
```

3 Distributive Laws

3.1 A Distributive Law for Causal Partial-Stream Functions

While the comonadic approach is quite powerful, there are natural notions of impure computation that it does not cover. One example is clocked dataflow or partial-stream based computation. The idea of clocked dataflow is that different signals may be on different clocks. Clocked dataflow signals can be represented by partial streams. A partial stream is a stream that may have empty positions to indicate the pace of the clock of a signal wrt. the base clock. The idea is to get rid of the different clocks by aligning all signals wrt. the base clock.

A very good news is that although comonads alone do not cover clocked dataflow computation, a solution is still close at hand. General and causal partial-stream functions turn out to be describable in terms of distributive combinations of a comonad and a monad considered, e.g., in [8,29]. For reasons of space, we will only discuss causal partial-stream functions as more relevant. General partial-stream functions are handled completely analogously.

Given a comonad $(D, \varepsilon, -^\dagger)$ and a monad $(T, \eta, -^\star)$ on a category $\mathcal{C}$, a *distributive law* of the former over the latter is a natural transformation λ with components $DTA \to TDA$ subject to four coherence conditions. A distributive law of D over T defines a category $\mathcal{C}_{D,T}$ where $|\mathcal{C}_{D,T}| = |\mathcal{C}|$, $\mathcal{C}_{D,T}(A, B) = \mathcal{C}(DA, TB)$, $(\mathsf{id}_{D,T})_A = \eta_A \circ \varepsilon_A$, $\ell \circ_{D,T} k = l^\star \circ \lambda_B \circ k^\dagger$ for $k : DA \to TB$, $\ell : DB \to TC$ (call it the *biKleisli category*), with inclusions to it from both the coKleisli category of D and Kleisli category of T.

In Haskell, the distributive combination is implemented as follows.

```
class (ComonadZip d, Monad t) => Dist d t where
  dist :: d (t a) -> t (d a)
```

The simplest examples of distributive laws are the distributivity of the identity comonad over any monad and the distributivity of any comonad over the identity monad.

```
instance Monad t => Dist Id t where
  dist (Id c) = mmap Id c

instance ComonadZip d => Dist d Id where
  dist d = Id (cmap unId d)
```

A more interesting example is the distributive law of the product comonad over the maybe monad.

```
data Maybe a = Just a | Nothing

instance Monad Maybe where
  return a      = Just a
  Just a  >>= k = k a
  Nothing >>= k = Nothing

instance Dist Prod Maybe where
  dist (Nothing :& _) = Nothing
  dist (Just a  :& e) = Just (a :& e)
```

For causal partial-stream functions, it is appropriate to combine the causal stream functions comonad LV with the maybe monad. And this is possible, since there is a distributive law which takes a partial list and a partial value (the past and present of the signal according to the base clock) and, depending on whether the partial value is undefined or defined, gives back the undefined list-value pair (the present time does not exist according to the signal's own clock) or a defined list-value pair, where the list is obtained from the partial list by leaving out its undefined elements (the past and present of the signal according to its own clock). In Haskell, this distributive law is coded as follows.

```
filterL :: List (Maybe a) -> List a
filterL Nil              = Nil
filterL (az :> Nothing) = filterL az
filterL (az :> Just a)  = filterL az :> a

instance Dist LV Maybe where
  dist (az := Nothing) = Nothing
  dist (az := Just a)  = Just (filterL az := a)
```

The biKleisli arrows of the distributive law are interpreted as partial-stream functions as follows.

```
runLVM  :: (LV  a -> Maybe b) -> Stream (Maybe a) -> Stream (Maybe b)
runLVM  k (a' :< as') = runLVM' k Nil a' as'

runLVM' k az Nothing  (a' :< as') = Nothing      :< runLVM' k az        a' as'
runLVM' k az (Just a) (a' :< as') = k (az := a)  :< runLVM' k (az :> a) a' as'
```

3.2 Distributivity-Based Semantics

Just as with comonads, we demonstrate distributive laws in action by presenting an interpreter. This time this is an interpreter of languages featuring both context-dependence and effects.

As previously, our first step is to fix the syntax of the object language.

```
type Var = String

data Tm = V Var | L Var Tm | Tm :@ Tm | Rec Tm
        | N Integer | Tm :+ Tm | ... | Tm :== Tm | ... | TT | FF | ... | If Tm Tm Tm
        | Tm `Fby` Tm                -- specific for causal stream functions
        | Nosig | Tm `Merge` Tm      -- specific for partiality
```

In the partiality part, Nosig corresponds to a nowhere defined stream, i.e., a signal on an infinitely slow clock. The function of Merge is to combine two partial streams into one which is defined wherever at least one of the given partial streams is defined.

The semantic domains and environments are defined as before, except that functions are now biKleisli functions, i.e., they take contextually situated values to values with an effect.

```
data Val d t = I Integer | B Bool | F (d (Val d t) -> t (Val d t))

type Env d t = [(Var, Val d t)]
```

Evaluation sends terms to biKleisli arrows; closed terms are interpreted in the empty environment placed into a context of interest.

```
class Dist d t => DistEv d t where
  ev :: Tm -> d (Env d t) -> t (Val d t)

evClosedLV :: DistEv LV t => Tm -> Int -> t (Val LV t)
evClosedLV e i = ev e (emptyL i := empty)
```

The meaning of the core constructs are essentially dictated by the types.

```
_ev :: DistEv d t => Tm -> d (Env d t) -> t (Val d t)
_ev (V x)        denv = return (unsafeLookup x (counit denv))
_ev (L x e)      denv = return (F (\ d -> ev e (cmap repair (czip d denv))))
                                where repair (a, env) = update x a env
_ev (e :@ e')    denv = ev e denv >>= \ (F f) ->
                        dist (cobind (ev e') denv) >>= \ d ->
                        f d
_ev (Rec e)      denv = ev e denv >>= \ (F f) ->
                        dist (cobind (_ev (Rec e)) denv) >>= \ d ->
                        f d
_ev (N n)        denv = return (I n)
_ev (e0 :+ e1)   denv = ev e0 denv >>= \ (I n0) ->
                        ev e1 denv >>= \ (I n1) ->
                        return (I (n0 + n1))
...
_ev TT           denv = return (B True )
_ev FF           denv = return (B False)
_ev (If e e0 e1) denv = ev e denv >>= \ (B b) -> if b then ev e0 denv else ev e1 denv
```

In monadic interpretation, the Rec operator is problematic, because recursive calls get evaluated too eagerly. The solution is to equip monads with a specific monadic fixpoint combinator mfix by making them instances of a type constructor class MonadFix (from Control.Monad.Fix). The same problem occurs here and is remedied by introducing a type constructor class DistCheat with a member function cobindCheat. The distributive law of LV over Maybe is an instance.

```
class Dist d t => DistCheat d t where
  cobindCheat :: (d a -> t b) -> (d a -> d (t b))

instance DistCheat LV Maybe where
  cobindCheat k d@(az := _) = cobindL k az := return (unJust (k d))
                     where cobindL k Nil = Nil
                           cobindL k (az :> a) = cobindL k az :> k (az := a)
```

Using the operation of the DistCheat class, the meaning of Rec can be redefined to yield a working solution.

```
class DistCheat d t => DistEv d t where ...

_ev (Rec e) denv = ev e denv >>= \ (F f) ->
                   dist (cobindCheat (_ev (Rec e)) denv) >>= \ d->
                   f d
```

The meanings of the constructs specific to the extension are also dictated by the types and here we can and must of course use the specific operations of the particular comonad and monad.

```
instance DistEv LV Maybe where
  ev (e0 ‘Fby‘ e1)   denv = ev e0 denv ‘fbyLV‘ cobind (ev e1) denv
  ev Nosig           denv = raise
  ev (e0 ‘Merge‘ e1) denv = ev e0 denv ‘handle‘ ev e1 denv
```

Partiality makes it possible to define a version of the sieve of Eratosthenes even in the causal setting (recall that our previous version without partiality used anticipation).

```
-- sieve x = if (tt fby ff) then x else sieve (if (x mod ini x /= 0) then x else nosig)
sieve = Rec (L "sieve" (L "x" (
               If (TT ‘Fby‘ FF)
                  (V "x")
                  (V "sieve" :@ (If ((V "x" ‘Mod‘ (ini :@ V "x")) :/= N 0) (V "x") Nosig)))))
-- eratosthenes = sieve (pos + 2)
eratosthenes = sieve :@ (pos :+ N 2)
```

Indeed, testing the above program, we get exactly what we would wish.

```
> runLVM (ev eratosthenes) (cmap Just emptyS)
Just 2 :< (Just 3 :< (Nothing :< (Just 5 :< (Nothing :< (Just 7 :< (Nothing :< (Nothing :< (
Nothing :< (Just 11 :< (Nothing :< (Just 13 :< (Nothing :< (Nothing :< (Nothing :< ...
```

4 Related Work

Semantic studies of Lucid, Lustre and Lucid Synchrone-like languages are not many and concentrate largely on the so-called clock calculus for static well-clockedness checking [9,10,13]. Relevantly for us, however, Colaço et al. [12] have very recently proposed a higher-order synchronous dataflow language extending Lucid Synchrone, with two type constructors of function spaces.

Hughes's arrows [17] have been picked up very well by the functional programming community (for overviews, see [26,18]). There exists by now not only a de facto standardized arrow library in Haskell, but even specialized syntax [25]. The main application is functional reactive programming with its specializations to animation, robotics etc. [23,16]. Functional reactive programming is continuous-time event-based dataflow programming.

Uses of comonads to structure notions of computation have been very few. Brookes and Geva [8] were the first to suggest this application. Kieburtz [19] made an attempt to draw the attention of functional programmers to comonads. Lewis et al. [21] must have contemplated employing the product comonad to handle implicit parameters, but did not carry out the project. Comonads have also been used in the semantics of intuitionistic linear logic and modal logics [5,7], with their applications in staged computation and elsewhere, see e.g., [14], and to analyse structured recursion schemes, see e.g., [34,24]. In the semantics of intuitionistic linear and modal logics, comonads are symmetric monoidal.

Our comonadic approach to stream-based programming is, to the best of our knowledge, entirely new. This is surprising, given how elementary it is. Workers in dataflow languages have produced a number of papers exploiting the final coalgebraic structure of streams [11,20,4], but apparently nothing on stream functions and comonads. The same is true about works in universal coalgebra [30,31].

5 Conclusions and Future Work

We have shown that notions of dataflow computation can be structured by suitable comonads, thus reinforcing the old idea that one should be able to use comonads to structure notions of context-dependent computation. We have demonstrated that the approach is fruitful with generic comonadic and distributivity-based interpreters that effectively suggest designs of dataflow languages. This is thanks to the rich structure present in comonads and distributive laws which essentially forces many design decisions (compare this to the much weaker structure in arrow types). Remarkably, the language designs that these interpreters suggest either coincide with the designs known from the dataflow languages literature or improve on them (when it comes to higher-orderness or to the choice of the primitive constructs in the case of clocked dataflow). For us, this is a solid proof of the true essence and structure of dataflow computation lying in comonads.

For future work, we envisage the following directions, in each of which we have already taken the first steps. First, we wish to obtain a solid understanding of the mathematical properties of our comonadic and distributivity-based semantics. Second, we plan to look at guarded recursion schemes associated to the comonads for stream functions and at language designs based on corresponding constructs. Third, we plan to test our interpreters on other comonads (e.g., decorated tree types) and see if they yield useful computation paradigms and language designs. Fourth, we also intend to study the pragmatics of the combination of two comonads via a distributive law. We believe that this will among other things explicate the underlying enabling structure of language designs such Multidimensional Lucid [3] where flows are multidimensional arrays. Fifth, the interpreters we have provided have been designed as reference specifications of language semantics. As implementations, they are grossly inefficient because of careless use of recursion, and we plan to investigate systematic efficient implementation of the languages they specify based on interpreter transformations. Sixth, we intend to take a close look at continuous-time event-based dataflow computation.

Acknowledgments. We are grateful to Neil Ghani for his suggestion to also look into distributive laws. This work was partially supported by the Estonian Science Foundation grant No. 5567.

References

1. P. Aczel, J. Adámek, S. Milius, J. Velebil. Infinite trees and completely iterative theories: A coalgebraic view. *Theoret. Comput. Sci.*, 300 (1–3), pp. 1–45, 2003.
2. E. A. Ashcroft, W. W. Wadge. *LUCID, The Dataflow Programming Language.* Academic Press, New York, 1985.
3. E. A. Ashcroft, A. A. Faustini, R. Jagannathan, W. W. Wadge. *Multidimensional Programming.* Oxford University Press, New York, 1995.
4. B. Barbier. Solving stream equation systems. In *Actes 13mes Journées Francophones des Langages Applicatifs, JFLA 2002*, pp. 117–139. 2002.
5. N. Benton, G. Bierman, V. de Paiva, M. Hyland. Linear lambda-calculus and categorical models revisited. In E. Börger et al., eds, *Proc. of 6th Wksh. on Computer Science Logic, CSL '92*, v. 702 of *Lect. Notes in Comput. Sci.*, pp. 61–84. Springer-Verlag, 1993.
6. N. Benton, J. Hughes, E. Moggi. Monads and effects. In G. Barthe, P. Dybjer, L. Pinto, J. Saraiva, eds., *Advanced Lectures from Int. Summer School on Applied Semantics, APPSEM 2000*, v. 2395 of *Lect. Notes in Comput. Sci.*, pp. 42–122. Springer-Verlag, Berlin, 2002.
7. G. Bierman, V. de Paiva. On an intuitionistic modal logic. *Studia Logica*, 65(3), pp. 383–416, 2000.
8. S. Brookes, S. Geva. Computational comonads and intensional semantics. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, eds., *Applications of Categories in Computer Science*, v. 177 of *London Math. Society Lecture Note Series*, pp. 1–44. Cambridge Univ. Press, Cambridge, 1992.
9. P. Caspi. Clocks in dataflow languages. *Theoret. Comput. Sci.*, 94(1), pp. 125–140, 1992.
10. P. Caspi, M. Pouzet. Synchronous Kahn networks. In *Proc. of 1st ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'96*, pp. 226–238. ACM Press, New York, 1996. Also in *SIGPLAN Notices*, 31(6), pp. 226–238, 1996.
11. P. Caspi, M. Pouzet. A co-iterative characterization of synchronous stream functions. In B. Jacobs, L. Moss, H. Reichel, J. Rutten, eds., *Proc. of 1st Wksh. on Coalgebraic Methods in Computer Science, CMCS'98*, v. 11 of *Electron. Notes in Theoret. Comput. Sci.*. Elsevier, Amsterdam, 1998.
12. J.-L. Colaço, A. Girault, G. Hamon, M. Pouzet. Towards a higher-order synchronous data-flow language. In *Proc. of 4th ACM Int. Conf. on Embedded Software, EMSOFT'04*, pp. 230–239. ACM Press, New York, 2004.
13. J.-L. Colaço, M. Pouzet. Clocks and first class abstract types. In R. Alur, I. Lee, eds., *Proc. of 3rd Int. Conf. on Embedded Software, EMSOFT 2003*, v. 2855 of *Lect. Notes in Comput. Sci.*, pp. 134–155. Springer-Verlag, Berlin, 2003.
14. R. Davies, F. Pfenning. A modal analysis of staged computation. *J. of ACM*, 48(3), pp. 555-604, 2001.
15. N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. The synchronous data flow programming language LUSTRE. *Proc. of the IEEE*, 79(9), pp. 1305–1320, 1991.
16. P. Hudak, A. Courtney, H. Nilsson, J. Peterson. Arrows, robots, and functional programming. In J. Jeuring, S. Peyton Jones, eds., *Revised Lectures from 4th Int. School on Advanced Functional Programming, AFP 2002*, v. 2638 of *Lect. Notes in Comput. Sci.*, pp. 159–187. Springer-Verlag, Berlin, 2003.
17. J. Hughes. Generalising monads to arrows. *Sci. of Comput. Program.*, 37(1–3), pp. 67–111, 2000.

18. J. Hughes. Programming with arrows. In V. Vene, T. Uustalu, eds., *Revised Lectures from 5th Int. School on Advanced Functional Programming, AFP 2004*, v. 3622 of *Lect. Notes in Comput. Sci.*, pp. 73–129. Springer-Verlag, Berlin, 2005.
19. R. B. Kieburtz. Codata and comonads in Haskell. Unpublished manuscript, 1999.
20. R. B. Kieburtz. Coalgebraic techniques for reactive functional programming, In *Actes 11mes Journées Francophones des Langages Applicatifs, JFLA 2000*, pp. 131–157. 2000.
21. J. R. Lewis, M. B. Shields, E. Meijer, J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *Proc. of 27th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'00*, pp. 108–118. ACM Press, New York, 2000.
22. E. Moggi. Notions of computation and monads. *Inform. and Comput.*, 93(1), pp. 55–92, 1991.
23. H. Nilsson, A. Courtney, J. Peterson. Functional reactive programming, continued. In *Proc. of 2002 ACM SIGPLAN Wksh. on Haskell, Haskell'02*, pp. 51–64. ACM Press, New York, 2002.
24. A. Pardo. Generic accumulations. J. Gibbons, J. Jeuring, eds., *Proc. of IFIP TC2/WG2.1 Working Conference on Generic Programming*, v. 243 of *IFIP Conf. Proc.*, pp. 49–78. Kluwer, Dordrecht, 2003.
25. R. Paterson. A new notation for arrows. In *Proc. of 6th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'01*, ACM Press, New York, pp. 229–240, 2001. Also in *SIGPLAN Notices*, 36(10), pp. 229–240, 2001.
26. R. Paterson. Arrows and computation. In J. Gibbons, O. de Moor, eds., *The Fun of Programming*, *Cornerstones of Computing*, pp. 201–222. Palgrave Macillan, Basingstoke / New York, 2003.
27. M. Pouzet. Lucid Synchrone: tutorial and reference manual. Unpublished manuscript, 2001.
28. J. Power, E. Robinson. Premonoidal categories and notions of computation. *Math. Structures in Comput. Sci.*, 7(5), pp. 453–468, 1997.
29. J. Power, H. Watanabe. Combining a monad and a comonad. *Theoret. Comput. Sci.*, 280(1–2), pp. 137–162, 2002.
30. J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoret. Comput. Sci.*, 249(1), pp. 3–80, 2000.
31. J. J. M. M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theoret. Comput. Sci.*, 308(1–3), pp. 1–53, 2003.
32. T. Uustalu, V. Vene. The dual of substitution is redecoration. In K. Hammond, S. Curtis (Eds.), *Trends in Functional Programming 3*, pp. 99–110. Intellect, Bristol / Portland, OR, 2002.
33. T. Uustalu, V. Vene. Signals and comonads. In M. A. Musicante, R. M. F. Lima, ed., *Proc. of 9th Brazilian Symp. on Programming Languages, SBLP 2005*, pp. 215–228. Univ. de Pernambuco, Recife, PE, 2005.
34. T. Uustalu, V. Vene, A. Pardo. Recursion schemes from comonads. *Nordic J. of Computing*, 8(3), pp. 366–390, 2001.
35. P. Wadler. The essence of functional programming. In *Conf. Record of 19th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'92*, pp. 1–14. ACM Press, New York, 1992.

Data Refinement with Low-Level Pointer Operations

Ivana Mijajlović[1] and Hongseok Yang[2,⋆]

[1] Queen Mary, University of London, UK
[2] ERC-ACI, Seoul National University, South Korea

Abstract. We present a method for proving data refinement in the presence of low-level pointer operations, such as memory allocation and deallocation, and pointer arithmetic. Surprisingly, none of the existing methods for data refinement, including those specifically designed for pointers, are sound in the presence of low-level pointer operations. The reason is that the low-level pointer operations allow an additional potential for obtaining the information about the implementation details of the module: using memory allocation and pointer comparison, a client of a module can find out which cells are internally used by the module, even without dereferencing any pointers. The unsoundness of the existing methods comes from the failure of handling this potential. In the paper, we propose a novel method for proving data refinement, called power simulation, and show that power simulation is sound even with low-level pointer operations.

1 Introduction

Data refinement [7] is a process in which the concrete representation of some abstract module is formally derived. Viewed from outside, the more concrete representation behaves the same as (or better than) the given abstract module. Thus, data refinement ensures that for every program, we can replace a given abstract module by the concrete one, while preserving (or even improving) the observable behavior of the program.

Our aim here is to develop a method of data refinement in the presence of *low-level* pointer operations, such as memory allocation and deallocation, and pointer arithmetic. Developing such methods is challenging, because low-level pointer operations allow subtle ways for accessing the internals of a module; without protecting the module internals from these accessing mechanisms (and thus ensuring that the module internals are only accessed by the module operations), we cannot have a sound method of data refinement.

The best-known accessing mechanism is the dereference of cross-boundary pointers. If a client program knows the location of some internal heap cell of a module, which we call a *cross-boundary pointer*, it can directly read or write that

⋆ Yang was supported by R08-2003-000-10370-0 from the Basic Research Program of Korea Science & Engineering Foundation, and Brain Korea 21 project in 2005.

```
module counter1 {                            module counter2 {          module counter3 {
  init() {*1=allocCell2(); **1=0;}             init() {*1=0;}             init() {*1=alloc(); **1=0;}
  inc() {**1=(**1)+1;}                         inc() {*1=(*1)+1;}         inc() {**1=(**1)+1;}
  read() {*3=(**1);}                           read() {*3=(*1);}          read() {*3=(**1);}
  final() {free(*1); *1=0;} }                  final() {*1=0;} }          final() {free(*1); *1=0;} }
```

Fig. 1. Counter Modules

internal cell by dereferencing that location. Thus, such a program can detect the changes in the representation of a module, and invalidate the standard methods of data refinements. This problem of cross-boundary pointers is well known, and several methods for data refinement have been proposed specifically to solve this problem [12, 17, 1, 3, 2].

However, none of the existing data-refinement methods, including the ones designed for cross-boundary pointers, can handle another accessing mechanism, which we call *allocation-status testing.* This mechanism uses the memory allocator and pointer comparison (with specific integers) to find out which cells are used internally by a module. A representative example check2 that implements this mechanism is z=alloc(); if (z==2) then v=1 else v=2. Assume that the memory allocator alloc nondeterministically chooses one inactive cell, and allocates the chosen cell. Under this assumption, check2 can detect whether cell 2 is used internally by a module or not. If a module is currently using cell 2, the newly allocated cell in check2 has to be different from 2, so that check2 always assigns 2 to v. On the other hand, if a module is not using cell 2, so cell 2 is free, then the memory allocation in check2 may or may not choose cell 2, and so, the variable v nondeterministically has value 1 or 2. Thus, by changing its nondeterministic behavior, check2 "observes" the allocation status of cell 2.

Protecting the module internals from the allocation-status testing is crucial for sound data refinement; using the allocation-status testing, a client can detect space-optimizing data refinements. We explain the issue with the first two counter modules, counter1 and counter2, in Fig. 1. Both modules implement a counter "object" with operations for incrementing the counter (inc) or reading the value of the counter (read). The main difference is that the second module uses less space than the first module. Let allocCell2() be a memory allocator that always selects cell 2: if cell 2 is inactive, allocCell2() allocates the cell; otherwise, i.e., if 2 is already allocated, then allocCell2() diverges. The first module is initialized by allocating cell 2 (allocCell2()) and storing the value of the counter in the allocated cell 2. The address of this newly allocated cell, namely 2, is kept in cell 1. On the other hand, the second module uses only cell 1, and stores the counter value directly to cell 1. The space-saving optimization in counter2 can be detected by the command check2 in the previous paragraph. When check2 is run with counter1, it always assigns 1 to v, but when check2 is run with the other module counter2, it can nondeterministically assign 1 or 2 to v. Thus, the optimization in counter2 is not correct, because it generates a new behavior of the client program check2.

Here, we present a data-refinement method that handles both cross-boundary pointers and allocation-status testing. Our method is based on Mijajlović *et al.*'s technique [12], which ensures correct data refinement in the presence of cross-boundary pointers, but as stressed there, not with allocation status testing. We provide a more general method which can cope well with both problems. The key idea of our method is to restrict the space optimization of a concrete module to nondeterministically allocated cells only, in order to hide the identities of the optimized cells from a client program, by making all the allocation-status testing fail to give any useful information. For instance, our method allows counter3 in Fig. 1 to be optimized by counter2, because the internal cell in counter3 is allocated nondeterministically. Note that even with check2, a client cannot detect this optimization, e.g. when cell 2 is free initially, counter3.init(); detect2 nondeterministically assigns 1 or 2 to v, just as counter2.init(); detect2 does. The precise formulation of our method uses a new notion of simulation – *power simulation*, to express this restriction on space optimization.

Related Work and Motivation. It has long been known that pointers cause great difficulties in the treatment of data abstraction [8, 9], and this has lead on to a non-trivial body of research [1, 3, 14, 11, 19, 17]. The focus of the present work (and [12]), on problems caused by low-level operations, sets it apart from all this other research.

Now, the reader might think that these problems arise only because of language bugs. Indeed, previous work has relied strongly on protection mechanisms of high-level, garbage collected languages. In such high-level languages, the nondeterministic memory allocation is harmless; it does not let one implement the allocation-status testing (because those languages forbid explicit deallocation and pointer arithmetic) and the nondeterministic allocation can even be treated deterministically using location renaming [19, 17]. Moreover, those high-level languages often have sophisticated type systems [3, 2] that limit cross-boundary pointers. However, we would counter that a comprehensive approach to abstraction cannot be based on linguistic restrictions. For, the fact of the existence of significant suites of infrastructure code – operating systems, database servers, network servers – argues against it. The architecture of this code is not enforced by linguistic mechanisms, and it is hard to see how it could be. Low-level code naturally uses cross-boundary pointers and address arithmetic. But it is a mistake to think that infrastructure code is unstructured; it often exhibits a large degree of pre-formal modularity. In this paper, we will demonstrate that there is no inherent reason why the *idea* of refinement of modules should not be applicable to it.

Outline. We start the paper by defining the storage model and the programming language in Sec. 2 and 3. Then, in Sec. 4, we describe the problem of finding a sound data-refinement method, and show that the usual forward method of data refinement fails to be a solution for the problem. In Sec. 5, we introduce the notion of *power simulation*, and prove its soundness; so, power simulation is a solution for the problem. Finally, in Sec. 6, we conclude the paper. The missing proofs of lemmas and propositions appear in the full version of the paper [13].

2 Storage Model and Finite Local Action

Our storage model, St, is the RAM model in separation logic [18, 10]:

$$\mathsf{Loc} = \{1, 2, \ldots\} \qquad \mathsf{Int} = \{\ldots, -2, -1, 0, 1, \ldots\} \qquad \mathsf{St} = \mathsf{Loc} \rightharpoonup_{\mathit{fin}} \mathsf{Int}$$

A state $h \in \mathsf{St}$ in the model is a finite mapping from locations to integer values; the domain of h denotes the set of currently allocated memory cells, and the "action" of h the contents of those allocated cells. Note that addresses are positive natural numbers, and so, they can be manipulated by arithmetic operations. We recall the disjointness predicate $h\#h'$ and the (partial) heap combining operator $h{\cdot}h'$ from separation logic. The predicate $h\#h'$ means that $\mathsf{dom}(h){\cap}\mathsf{dom}(h') \neq \emptyset$; and, $h{\cdot}h'$ is defined only for such disjoint heaps h and h', and in that case, it denotes the combined heap $h \cup h'$. We overload the disjointness predicate $\#$, and for states h and location sets L, we write $h\#L$ to mean that all locations in L are free in h (i.e., $\mathsf{dom}(h) \cap L = \emptyset$).

We specify a property of storage, using subsets of St directly, instead of syntactic formulas. We call such subsets of St *predicates*, and use semantic versions of separating conjunction $*$ and preciseness from separation logic:

$$p, q \in \mathsf{Pred} \overset{\mathit{def}}{=} \wp(\mathsf{St}) \qquad p * q \overset{\mathit{def}}{=} \{h_p{\cdot}h_q \mid h_p \in p \wedge h_q \in q\} \qquad \mathsf{true} \overset{\mathit{def}}{=} \mathsf{St}$$
$$p \text{ is precise} \overset{\mathit{def}}{\Leftrightarrow} \text{for all } h\text{, there is at most one splitting } h_p{\cdot}h_0{=}h \text{ of } h \text{ s.t. } h_p{\in}p.$$

An *action* r is a relation from St to $\mathsf{St} \cup \{\mathsf{av}, \mathsf{flt}\}$. Intuitively, it denotes a nondeterministic client program that uses a module. Action r can output two types of errors, access violation av and memory fault flt. The first error av means that a client attempts to break the boundary between the client and the module, by accessing the internals of the module directly without using module operations. The second one, flt, means that a client tries to dereference a null or a dangling pointer. Note that if $\neg h[r]\mathsf{flt}$, state h contains all the cells that r dereferences, except the newly allocated cells. As in separation logic, we write $\mathsf{safe}(r, h)$ to indicate this (i.e., $\neg h[r]\mathsf{flt}$).

A *finite local action* is an action that satisfies: *safety monotonicity*, *frame property*, *finite access property*, and *contents independence*. Intuitively, these four properties mean that each execution of the action accesses only finitely many heap cells. Some of the cells are accessed directly by pointer dereferencing, so that the contents of the cells affects the execution, while the other remaining cells are accessed only indirectly by the allocation-status testing, so that the execution only depends on the allocation status of the cells, not their contents. More precisely, we define the four properties as follows:[1]

– **Safety Monotonicity**: if $h_0\#h_1$ and $\mathsf{safe}(r, h_0)$, then $\mathsf{safe}(r, h_0{\cdot}h_1)$.
– **Frame Property**: if $\mathsf{safe}(r, h_0)$ and $h_0{\cdot}h_1[r]h'$, then $\exists h_0'.\ h'{=}h_0'{\cdot}h_1 \wedge h_0[r]h_0'$.

[1] All the states free in the properties are universally quantified.

– **Finite Access Property**: if $\mathsf{safe}(r, h_0)$ and $h_0[r]h_0'$, then $\exists L{\subseteq_{\mathit{fin}}}\mathsf{Loc}.\,\forall h_1.\,(h_1\#h_0 \,\wedge\, h_1\#h_0' \,\wedge\, (\mathsf{dom}(h_1) \cap L = \emptyset)) \;\Rightarrow\; h_0{\cdot}h_1[r]h_0'{\cdot}h_1$.
– **Contents Independence:** if $\mathsf{safe}(r, h_0)$ and $h_0{\cdot}h_1[r]h_0'{\cdot}h_1$, then $h_0{\cdot}h_2[r]h_0'{\cdot}h_2$ for all states h_2 with $\mathsf{dom}(h_1){=}\mathsf{dom}(h_2)$.

The first two properties are well-known locality properties from separation logic, and mean that if h_0 contains all the directly accessed cells by a "command" r, every computation from a bigger state $h_0{\cdot}h_1$ is safe, and it can be tracked by some computation from the smaller state h_0. The third condition expresses the converse; every computation from the smaller state h_0 can be extended to a computation from the bigger state $h_0{\cdot}h_1$, as long as the extended part h_1 does not include directly accessed locations ($h_1\#h_0 \wedge h_1\#h_0'$) or indirectly accessed locations (i.e., $\mathsf{dom}(h_1) \cap L = \emptyset$). Note that the finite set L contains all the indirectly accessed locations by the computation $h_0[r]h_0'$. The last one, contents independence, expresses that if $\mathsf{safe}(r, h_0)$, the execution of r from a bigger state $h_0{\cdot}h_1$ does not look at the contents of cells in h_1; it can only use the information that the locations in h_1 are allocated initially. At first glance, it may seem that contents independence follows from the frame property, but the following example suggests otherwise. Let $[]$ be the empty state, and let r be an action defined by $h[r]v \Leftrightarrow h = v \,\wedge\, (h = [] \,\vee\, (1{\in}\mathsf{dom}(h) \wedge h(1){=}2))$. This "command" r satisfies both the safety monotonicity and the frame property, but not the contents independence; even though $\mathsf{safe}(r, [])$ and $1 \notin \mathsf{dom}([])$, "command" r behaves differently depending on the contents of cell 1. The finite access property and contents independence are new in this paper, and they play an important role in the soundness of our data-refinement method (Sect. 5.1).

Definition 1 (Finite Local Action). *A* finite local action*, in short* FLA*, is an action that satisfies safety monotonicity, frame property, finite access property, and contents independence. A finite local action is* av-free *iff it does not relate any state to* av*.*

The set of finite local actions has a structure rich enough to interpret programs with all the low-level pointer operations that have been considered in separation logic.[2] Let $\mathcal{F}$ be the poset of FLAs ordered by the "graph-subset" relation $\sqsubseteq$[3], and let $\mathcal{F}_{\mathit{noav}}$ be the sub-poset of $\mathcal{F}$ consisting of av-free FLAs. Particularly interesting are the low-level pointer operations, such as the memory update, allocation and deallocation of a cell, and a test "$*l \in I$" for location l and integer set I: if l is allocated and it contains a value in I, the test skips; if l is allocated but its value is not in I, the test blocks; otherwise (i.e., if l is not allocated), the test generates the memory fault flt. For instance, $\mathsf{test}(1, \{3\})$ expresses the conditional statement $\mathsf{if}\ (*1{\neq}3)\{\mathsf{diverge}\}$. Note that $\mathsf{test}(1, \{3\})$ generates flt precisely when the boolean condition $*1{\neq}3$ dereferences an inactive cell.

[2] Thus, the set of finite local actions, as a semantic domain, expresses the computational behavior of pointer programs more accurately than the set of local actions, just as the set of continuous functions is a more "accurate" semantic domain than that of monotone functions in the domain theory.

[3] $r \sqsubseteq r'$ iff $\forall h \in \mathsf{St}.\forall v \in \mathsf{St} \cup \{\mathsf{flt}, \mathsf{av}\}.\, h[r]v \Rightarrow h[r']v$.

Let l be a location, i an integer, n a positive natural number, and I a set of integers.

$h[\mathsf{update}(l,i)]v \overset{def}{\Leftrightarrow}$ if $l \notin \mathsf{dom}(h)$ then $v{=}\mathsf{flt}$ else $v{=}h[l \mapsto i]$

$h[\mathsf{cons}(l,n)]v \overset{def}{\Leftrightarrow}$ if $l \notin \mathsf{dom}(h)$ then $v{=}\mathsf{flt}$ else $(\exists l'.\ v{=}(h[l \mapsto l'])\cdot[l'{\to}0,..,l'{+}n{-}1{\to}0])$

$h[\mathsf{dispose}(l)]v \overset{def}{\Leftrightarrow}$ if $l \notin \mathsf{dom}(h)$ then $v{=}\mathsf{flt}$ else $v\cdot[l{\to}h(l)]{=}h$

$h[\mathsf{test}(l,I)]v \overset{def}{\Leftrightarrow}$ if $l \notin \mathsf{dom}(h)$ then $v{=}\mathsf{flt}$ else $(v{=}h \wedge h(l){\in}I)$

Fig. 2. Semantic Low-level Pointer Operations

Lemma 1. *The poset $\mathcal{F}_{noav}$ of* av*-free FLAs contains the operations in Fig. 2.*

Lemma 2. *Both $\mathcal{F}$ and $\mathcal{F}_{noav}$ are complete lattices that have the set union as their join operator: for every family $\{r_i\}_{i\in I}$ in each poset, $\bigsqcup_{i\in I} r_i$ is $\bigcup_{i\in I} r_i$.*

3 Programming Language

The programming language is Dijkstra's language of guarded commands [5] extended with low-level pointer operations and module operations. The syntax of the language is given by the grammar:

$$C ::= f \mid a \mid C;C \mid C[]C \mid P \mid \mathsf{fix}\,P.\,C$$

where f, a, P are, respectively, chosen from three disjoint sets mop, aop, pid of identifiers. The first construct f is a module operation declared in the "interface specification" mop. Before a command in our language gets executed, it is first "linked" to a specific module that implements the interface mop. This linked module provides the meaning of the command f. The second construct a is an atomic operation, which a client can execute without using the module operations. Usually, a denotes a low-level pointer operation. Note that the language does not provide a syntax for building specific pointer operations. Instead, we assume that the interpretation $[\![-]\!]_a$ of these atomic client operations as av-free FLAs is given along with aop, and that under this interpretation, aop includes at least all the pointer operations in Lemma 1, so that aop includes all the atomic pointer operations considered in separation logic. The remaining four constructs of the language are the usual compound commands from Dijkstra's language: sequential composition $C;C$, nondeterministic choice $C[]C$, the call of a parameterless procedure P, and the recursive definition $\mathsf{fix}\,P.\,C$ of a parameterless procedure. As in Dijkstra's language, the construct $\mathsf{fix}\,P.\,C$ not only defines a parameterless recursive procedure P, but also calls the defined procedure. We express that a command C does not have free procedure names, by calling C a *complete command.*

We interpret commands using an instrumented denotational semantics; besides computing the usual state transformation, the semantics also checks whether each atomic client operation accesses the internals of a module, and for such illegal accesses, the semantics generates an access violation av.

To implement the instrumentation, we parameterize the semantics by what we call a *semantic module.* Let init and final be identifiers that are not in mop. A

$$\mu \in \mathcal{E} \overset{def}{=} \mathsf{pid} \to \mathcal{F} \qquad [\![C]\!]_{(p,\eta)} : \mathcal{E} \to \mathcal{F} \qquad [\![C]\!]^c_{(p,\eta)} : \mathcal{F} \ \text{(for complete } C\text{)}$$

$$[\![a]\!]_{(p,\eta)}\mu \overset{def}{=} \mathsf{prot}([\![a]\!]_a, p) \qquad [\![C[\!]C']\!]_{(p,\eta)}\mu \overset{def}{=} [\![C]\!]_{(p,\eta)}\mu \cup [\![C]\!]_{(p,\eta)}\mu$$

$$[\![f]\!]_{(p,\eta)}\mu \overset{def}{=} \eta(f) \qquad [\![P]\!]_{(p,\eta)}\mu \overset{def}{=} \mu(P)$$

$$[\![C;C']\!]_{(p,\eta)}\mu \overset{def}{=} \mathsf{seq}([\![C]\!]_{(p,\eta)}\mu, [\![C']\!]_{(p,\eta)}\mu) \qquad [\![\mathsf{fix}\,P.\,C]\!]_{(p,\eta)}\mu \overset{def}{=} \mathsf{fix}\ \lambda r.\, [\![C]\!]_{(p,\eta)}(\mu[P{\to}r])$$

$$[\![C]\!]^c_{(p,\eta)} \overset{def}{=} \mathsf{seq}(\mathsf{seq}(\eta(\mathsf{init}), [\![C]\!]_{(p,\eta)}\bot), \eta(\mathsf{final})) \ \text{(for complete } C\text{)}$$

where $\mathsf{seq}\colon \mathcal{F}{\times}\mathcal{F}{\to}\mathcal{F}$ and $\mathsf{prot}\colon \mathcal{F}{\times}\mathsf{Pred}{\to}\mathcal{F}$ are defined as follows:

$$h[\mathsf{prot}(r,p)]v \overset{def}{\Leftrightarrow} h[r]v \ \vee\ (v{=}\mathsf{av} \ \wedge\ \neg h[r]\mathsf{flt} \ \wedge\ \exists h_p, h_0.\, h{=}h_p{\cdot}h_0 \wedge h_p \in p \wedge h_0[r]\mathsf{flt})$$

$$h[\mathsf{seq}(r,r')]v \overset{def}{\Leftrightarrow} (\exists h'.\, h[r]h' \ \wedge\ h'[r']v) \ \vee\ (h[r]\mathsf{flt} \ \wedge\ v{=}\mathsf{flt}) \ \vee\ (h[r]\mathsf{av} \ \wedge\ v{=}\mathsf{av})$$

Fig. 3. Semantics of Language

semantic module is a pair of a predicate p and a function η from $\mathsf{mop}\cup\{\mathsf{init},\mathsf{final}\}$ to $\mathcal{F}_{noav}$, such that (1) $\forall h,h'.\,(\mathsf{safe}(\eta(\mathsf{init}),h) \wedge h[\eta(\mathsf{init})]h' \Rightarrow h'{\in}p{*}\mathsf{true})$; (2) for all f in mop, $\forall h,h'.\,(\mathsf{safe}(\eta(f),h) \wedge h{\in}p{*}\mathsf{true} \wedge h[\eta(f)]h' \Rightarrow h'{\in}p{*}\mathsf{true})$; (3) p is precise. Intuitively, the predicate p in the semantic module denotes the resource invariant for the module internals, and function η specifies the meaning of the module operations, initialization init and finalization final. The first condition of the semantic module requires initialization to establish the resource invariant, and the second condition, that the established resource invariant be preserved by module operations. The last condition is more subtle. It ensures that using the invariant p, we can determine which part of each state belongs to the module. Recall that a predicate q is precise iff every state h in $q * \mathsf{true}$ has a unique splitting $h_q{\cdot}h_0 = h$ such that $h_q \in q$. Thus, if p is precise, then for every state h containing both the internals and externals of the module (i.e., $h \in p * \mathsf{true}$), we can unambiguously split h into module-owned part h_p and client-owned part h_0. This unambiguous splitting is used in the semantics to detect the access violation of the atomic client operations, and it also plays a crucial role in the soundness of our refinement method (Sect. 5.1). We remark that requiring the preciseness of the invariant p is not as restrictive as one might think, because most of the used resource invariants are precise; among the used resource invariants in separation logic, only one invariant is not precise, but even that invariant can safely be tightened to a precise one.[4]

Let $\mathcal{E}$ be the poset of all functions from pid to $\mathcal{F}$ ordered pointwise. Given semantic module (p,η), we interpret a command as a continuous function $[\![-]\!]_{(p,\eta)}$ from $\mathcal{E}$ to $\mathcal{F}$. For complete commands C, we consider an additional interpretation $[\![-]\!]^c_{(p,\eta)}$ that uses the least environment $\bot = \lambda P.\emptyset$, and runs the initialization and the finalization of the module (p,η) before and after $([\![C]\!]_{(p,\eta)}\bot)$, respectively. The details of these two interpretations are shown in Fig. 3.

The most interesting part of the semantics lies in the interpretation of the atomic client operations. For each atomic operation a, its interpretation first

[4] The only known unprecise invariant is $\mathsf{listseg}(x,y)$ in [18], which means the existence of a (possibly cyclic) linked list segment from x to y. However, even that invariant can be made precise, if it is restricted to forbid a cycle in the list segment [15].

looks up the original meaning $[\![a]\!]_a \in \mathcal{F}_{noav}$, which is given when the syntax of the language is defined. Then, the interpretation transforms the meaning into $\mathsf{prot}([\![a]\!]_a, p)$, "the p-protected execution of $[\![a]\!]_a$." Intuitively, $\mathsf{prot}([\![a]\!]_a, p)$ behaves the same as $[\![a]\!]_a$, except that whenever $[\![a]\!]_a$ accesses the p-part of the input state, $\mathsf{prot}([\![a]\!]_a, p)$ generates av, thus indicating that there is an "access violation." Since p is the resource invariant of the module, $\mathsf{prot}([\![a]\!]_a, p)$ notifies all illegal accesses to the module internals, by generating av.

Lemma 3. *The interpretation in Fig. 3 is well-defined.*

4 Data Refinement

The goal of this paper is to find a method for proving that a "concrete" module (q, ϵ) data-refines an "abstract" module (p, η). In this section, we first formalize this goal by defining the notion of data refinement. Then, we demonstrate the difficulty of achieving the goal, by showing that the standard forward method is not sound in the presence of allocation-status testing.

We use the notion of data refinement that Mijajlović *et al.* devised in order to handle cross-boundary pointers. Usually, data refinement is a relation between modules defined by substitutability: a module (q, ϵ) data-refines another module (p, η) iff for all complete commands C using (p, η), substituting the concrete module (q, ϵ) for the abstract module (p, η) improves the behavior of C, i.e., C becomes more deterministic with the concrete module. Mijajlović *et al.* weakened this usual notion of data refinement, by dropping the requirement about improvement for error-generating input states: if C with the abstract module (p, η) generates an access violation av or a memory fault flt from an input state h, then for this input h, the data refinement does not constrain the execution of C with the concrete module (q, ϵ), and allows it to generate any outputs. In this paper, we use the following formalization of this weaker notion of data refinement:

Definition 2 (Data Refinement). *A module (q, ϵ)* data-refines *another module (p, η) iff for all complete commands C and all states h, if $[\![C]\!]^c_{(p,\eta)}$ does not generate an error from h (i.e., $\neg h[[\![C]\!]^c_{(p,\eta)}]\mathsf{av} \wedge \neg h[[\![C]\!]^c_{(p,\eta)}]\mathsf{flt}$), then*

$$\big(\neg h[[\![C]\!]^c_{(q,\epsilon)}]\mathsf{av} \;\wedge\; \neg h[[\![C]\!]^c_{(q,\epsilon)}]\mathsf{flt}\big) \;\wedge\; \big(\forall h'.\; h[[\![C]\!]^c_{(q,\epsilon)}]h' \;\Rightarrow\; h[[\![C]\!]^c_{(p,\eta)}]h'\big).$$

The main benefit of considering this notion of data refinement is that a proof method for data refinement does not have to do anything special in order to handle the cross-boundary pointers. Recall that flt means that a command tries to dereference dangling pointers or nil, and av means that a command attempts to dereference the internal cells of a module without using module operations. Thus, if a command C does not generate an error from an input state h, then all the cells that C directly dereferences during execution must be allocated and belong to the "client" portion of the state; in particular, C does not dereference any cross-boundary pointers directly. Since the data refinement now asks for the improvement of only the error-free computations of C, a proof method for data refinement can ignore the "bad" computations where C dereferences cross-boundary pointers.

Unfortunately, even with this weaker notion of data refinement, standard proof methods for data refinement are not sound; they fail to deal with the allocation-status testing. We explain this soundness problem making use of the notion of the forward simulation in [12]. As pointed out in their work, while successfully dealing with the cross-boundary pointer dereferencing problem, the forward method is not sound for allocation-status testing.

The key concept of the forward simulation in [12] is an operator fsim that maps a pair (R_0, R_1) of state relations to a relation $\mathsf{fsim}(R_0, R_1)$ between FLAs. Intuitively, $r'[\mathsf{fsim}(R_0, R_1)]r$ means that given R_0-related input states h' and h, if r does not generate an error from h, then (1) r' does not generates an error from h' and (2) every output of r' from h' is R_1-related to some outcome of r. More precisely, $r'[\mathsf{fsim}(R_0, R_1)]r$ iff for all states h' and h, if $(h'[R_0]h \wedge \neg h[r]\mathsf{flt} \wedge \neg h[r]\mathsf{av})$, then

$$\big(\neg h'[r']\mathsf{flt} \ \wedge\ \neg h'[r']\mathsf{av}\big) \ \wedge\ \big(\forall h'_1.\, h'[r']h'_1 \ \Rightarrow\ \exists h_1.\, h[r]h_1 \wedge h'_1[R_1]h_1\big).$$

The condition about the absence of errors comes from the fact that the data refinement considers only error-free computations. Except this condition, the way of relating two actions (or commands) in $\mathsf{fsim}(R_0, R_1)$ is fairly standard in the work on data refinement [6, 4].

Let $\varDelta$ be the diagonal relation on states[5], and for state relations R_0 and R_1, let $R_0 * R_1$ be their relational separating conjunction [17]: $h'[R_0 * R_1]h$ iff h' and h are, respectively, split into $h'_0{\cdot}h'_1 = h'$ and $h_0{\cdot}h_1 = h$ such that the first parts h'_0, h_0 are related by R_0 and the second parts h'_1, h_1 by R_1.[6] The formal definition of forward simulation is given below:

Definition 3 (Forward Simulation). *Let $(q, \epsilon), (p, \eta)$ be semantic modules, and R a relation s.t. $R \subseteq q \times p$.* Module (q, ϵ) forward-simulates (p, η) by R *iff*

1. $\epsilon(\mathsf{init})[\mathsf{fsim}(\varDelta, R * \varDelta)]\eta(\mathsf{init})$ *and* $\epsilon(\mathsf{final})[\mathsf{fsim}(R * \varDelta, \varDelta)]\eta(\mathsf{final})$*;*
2. $\forall f \in \mathsf{mop}.\ \epsilon(f)[\mathsf{fsim}(R * \varDelta, R * \varDelta)]\eta(f)$.

The relation $R * \varDelta$ here expresses that the corresponding states of r' and r can, respectively, be partitioned into the module and client parts; the module parts of r' and r are related by R, but the client parts of r' and r are the same.

The forward simulation is not sound: there are modules $(q, \epsilon), (p, \eta)$ such that the concrete module (q, ϵ) forward-simulates the abstract module(p, η) by some $R \subseteq q \times p$, but it does not data-refine it. The main reason of this unsoundness is that the low-level pointer operations in our language, especially those implementing allocation-status testing, break the underlying assumption of the forward simulation. The forward simulation assumes a language where if a command C does not call module operations, then for all relations $R \subseteq q \times p$, the

[5] $h'[\varDelta]h \overset{def}{\Leftrightarrow} h' = h$.

[6] $h'[R_0 * R_1]h \overset{def}{\Leftrightarrow} \exists h'_0, h'_1, h_0, h_1.\ h'_0{\cdot}h'_1 = h' \ \wedge\ h_0{\cdot}h_1 = h \ \wedge\ h'_0[R_0]h_0 \ \wedge\ h'_1[R_1]h_1$.

command "forward-simulates" itself by R: $[\![C]\!]_{(q,\epsilon)}\mu'[\mathsf{fsim}(R{*}\Delta, R{*}\Delta)][\![C]\!]_{(p,\eta)}\mu$ for all μ', μ that define $\mathsf{fsim}(R{*}\Delta, R{*}\Delta)$-related "procedures". Our language, however, does not satisfy this assumption; if an atomic client command a implements the allocation-status testing, it is not related to itself by $\mathsf{fsim}(R{*}\Delta, R{*}\Delta)$ in general. For instance, having a concrete module (q, ϵ) and the abstract one (p, η) and a relation R between them, consider an atomic command $\mathsf{cons}(2, 1)$ that allocates one new cell initialized to 0 and assigns its address to cell 2; in case that cell 2 is not allocated initially, $\mathsf{cons}(2, 1)$ generates flt. [7] Let R be defined by $h_0'[R]h_0 \Leftrightarrow h_0' = [] \wedge h_0 = [1{\to}2]$. Then, $h'[R{*}\Delta]h$ iff there is some state h_1 such that $1 \notin \mathsf{dom}(h_1) \wedge h' = []{\cdot}h_1 \wedge h = [1{\to}2]{\cdot}h_1$. Thus, states $h' = [2{\to}0]$ and $h = [1{\to}2, 2{\to}0]$ are $R{*}\Delta$-related. We will now consider the execution of $\mathsf{cons}(2, 1)$ from these $R{*}\Delta$-related states h' and h. When $\mathsf{cons}(2, 1)$ is run from h' (with the concrete module (q, ϵ)), it can allocate cell 1 and give the output state $h_1' = [1{\to}0, 2{\to}1]$ (i.e., $h_1[[\![\mathsf{cons}(2, 1)]\!]_{(q,\epsilon)}\mu']h_1'$), because cell 1 is free initially (i.e., $1 \notin \mathsf{dom}(h')$). However, when the same command is run from h (with the abstract module (p, η)), it cannot allocate cell 1, because 1 is already active in h (i.e., $1 \in \mathsf{dom}(h)$). In this case, all the output states of $\mathsf{cons}(2, 1)$ have the form $[1{\to}0, 2{\to}n, n{\to}0]$ for some $n \in \mathsf{Nats}{-}\{1, 2\}$. Note that the state $h_1' = [1{\to}0, 2{\to}1]$ is not $R{*}\Delta$-related to any such outputs $[1{\to}0, 2{\to}n, n{\to}0]$. Thus, we cannot have that $\big([\![\mathsf{cons}(2, 1)]\!]_{(q,\epsilon)}\mu'\big)[\mathsf{fsim}(R{*}\Delta, R{*}\Delta)]\big([\![\mathsf{cons}(2, 1)]\!]_{(p,\eta)}\mu\big)$. In the full version of the paper [13], we have used these R and cons to construct a counter example for the soundness of the forward simulation.

5 Power Simulation

We now present the main result of this paper: a new method for data refinement, called power simulation, and its soundness proof.

The key idea of power simulation is to use the state-set lifting $\mathsf{lft}(r)$ of a FLA:

$$\begin{aligned} \mathsf{lft}(r) \ &: \ \wp(\mathsf{St}) \leftrightarrow (\wp(\mathsf{St}) \cup \{\mathsf{flt}, \mathsf{av}\}) \\ H[\mathsf{lft}(r)]V &\overset{def}{\Leftrightarrow} (V{\subseteq}\mathsf{St} \wedge \forall h'{\in}V. \exists h{\in}H.\, h[r]h') \vee ((V{=}\mathsf{av} \vee V{=}\mathsf{flt}) \wedge \exists h{\in}H.\, h[r]V). \end{aligned}$$

Given an input state set H, the "lifted command" $\mathsf{lft}(r)$ runs r for all the states in H, chooses some states among the results, and returns the set V of the chosen states. Note that V might not contain some possible outputs from H; so, $\mathsf{lft}(r)$ is different from the usual direct image map of r, and in general, it is a relation rather than a function. For each module (p, η), we write $\mathsf{lft}(\eta)$ for the lifting of all module operations (i.e., $\forall f \in \mathsf{mop}.\ \mathsf{lft}(\eta)(f) = \mathsf{lft}(\eta(f))$), and call $(p, \mathsf{lft}(\eta))$ the *lifting* of (p, η).

The power simulation is the usual forward simulation of a *lifted* "abstract" module by a *normal* "concrete" module. Suppose that we want to show that a concrete module (q, ϵ) data-refines an abstract module (p, η). Define a power relation to be a relation between states and state sets. Intuitively, the power simulation says that to prove this data refinement, we only need to find a "good"

[7] $h[[\![\mathsf{cons}(2, 1)]\!]_a]v \overset{def}{\Leftrightarrow}$ if $2 \notin \mathsf{dom}(h)$ then $v{=}\mathsf{flt}$else $\exists n.\, v{=}h[2{\to}n]{\cdot}[n{\to}0]$.

power relation $\mathcal{R} \subseteq \mathsf{St} \times \wp(\mathsf{St})$ such that every concrete-module operation $\epsilon(k)$ "forward-simulates" the corresponding lifted abstract-module operation $\mathsf{lft}(\eta(k))$ by $\mathcal{R}$. The official definition of power simulation formalizes this intuition by specifying (1) which power relation should be considered good for given modules (q, ϵ) and (p, η), and (2) what it means that a normal command "forward-simulates" a lifted command. For the first, we use the *expansion* operator and *admissibility* condition for power relations. For the second, we use the operator psim that maps a power-relation pair to a relation on FLAs. We will now define these subcomponents of power simulation, and use them to give the formal definition of power simulation.

We explain operator psim first. For power relations $\mathcal{R}_0$ and $\mathcal{R}_1$, $\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)$ relates a "concrete" FLA r' with an "abstract" r iff for every $\mathcal{R}_0$-related input state h' and state set H, if $\mathsf{lft}(r)$ does not generate an error from H, then all the outputs of r' from h' are $\mathcal{R}_1$-related to some output state sets of $\mathsf{lft}(r)$ from H. More precisely, $r'[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)]r$ iff for all h' and H, if $h'[\mathcal{R}_0]H$ and neither $H[\mathsf{lft}(r)]\mathsf{flt}$ nor $H[\mathsf{lft}(r)]\mathsf{av}$, then

$$\big(\neg h'[r']\mathsf{flt} \;\wedge\; \neg h'[r']\mathsf{av}\big) \;\wedge\; \big(\forall h'_1.\, h'[r']h'_1 \;\Rightarrow\; \exists H_1.\, H[\mathsf{lft}(r)]H_1 \wedge h'_1[\mathcal{R}_1]H_1\big).$$

Note that this definition is the lifted version of fsim in Sec. 4; except that it considers the lifted computation $\mathsf{lft}(r)$, instead of the usual computation r, it coincides with the definition of fsim. In the definition of power simulation, we will use this psim to express the "forward-simulation" of a lifted command by a normal command.

Next, we define the expansion operator $-\otimes\mathit{\Delta}$ for power relations. The expansion $\mathcal{R}\otimes\mathit{\Delta}$ of a power relation $\mathcal{R}$ is a power relation defined as follows:

$$h[\mathcal{R}\otimes\mathit{\Delta}]H \;\overset{def}{\Leftrightarrow}\; \exists h_r, h_0, H_r.\big(h = h_r{\cdot}h_0 \;\wedge\; h_r[\mathcal{R}]H_r \;\wedge\; H = H_r * \{h_0\}\big).$$

Intuitively, the definition means that h and H are obtained by extending $\mathcal{R}$-related state h_r and state sets H_r by the same state h_0. Usually, $\mathcal{R}$ is a "coupling" power relation that connects the internals of two modules, and $\mathcal{R}\otimes\mathit{\Delta}$ expands this coupling relation to the relation for the entire memory, by asking that the added client parts must be identical.

The final subcomponent of power simulation is the admissibility condition for power relations. A power relation $\mathcal{R}$ is *admissible* iff for every $\mathcal{R}$-related state h and state set H (i.e., $h[\mathcal{R}]H$), we have that[8]

$$H \neq \emptyset \;\wedge\; \big(\forall L{\subseteq_{\mathit{fin}}}\mathsf{Loc}{-}\mathsf{dom}(h).\, \exists H_1{\subseteq}H.\, \big(H_1{\neq}\emptyset \wedge h[\mathcal{R}]H_1 \wedge \forall h_1{\in}H_1.\, h_1\#L\big)\big).$$

The first conjunct in the admissibility condition means that all related state sets must contain at least one state. The second conjunct is about the "free cells" in these related state sets. It means that if $h[\mathcal{R}]H$, state set H collectively has at least as many free cells as h: for every finite collection L of free cells in h, set H contains states that do not have any of the cells in L, and, moreover, the set

[8] Recall that $h_1\#L$ iff $\mathsf{dom}(h_1) \cap L = \emptyset$.

H_1 of such states itself collectively has as many free cells as h. To understand the second conjunct more clearly, consider power relations $\mathcal{R}_0, \mathcal{R}_1, \mathcal{R}_2$ defined as follows:

$$h[\mathcal{R}_0]H \overset{def}{\Leftrightarrow} h{=}[3{\to}1] \wedge H{=}\{[3{\to}5]\} \qquad h[\mathcal{R}_1]H \overset{def}{\Leftrightarrow} h{=}[3{\to}1] \wedge H{=}\{[3{\to}5, 4{\to}5]\}$$
$$h[\mathcal{R}_2]H \overset{def}{\Leftrightarrow} h{=}[3{\to}1] \wedge \exists L{\subseteq_{fin}}\mathsf{Loc}.\ H{=}\{[3{\to}5, n{\to}5] \mid n \notin L \cup \{3\}\}$$

The first power relation $\mathcal{R}_0$ is admissible, because set $\{[3{\to}5]\}$ has only one state $[3{\to}5]$ that has the exactly same free cells, namely all cells other than 3, as state $[3{\to}1]$. On the other hand, $\mathcal{R}_1$ is not admissible, because the (unique) state in $\{[3{\to}5, 4{\to}5]\}$ has an active cell 4 that is not free in $[3{\to}1]$. The last relation $\mathcal{R}_2$ is tricky; relation $\mathcal{R}_2$ is admissible, even though for all $\mathcal{R}_2$-related h and H, every state in H has more active cells than h. The intuitive reason for this is that for every free cell in $[3{\to}1]$, set H contains a state that does not contain the cell, and so, it collectively has as many free cells as $[3{\to}1]$; in a sense, by having sufficiently many states, H hides the identity of the additional cell n. The formal proof that $\mathcal{R}_2$ satisfies the second conjunct of the admissibility condition proceeds as follows. Consider H, h', L_1 such that $h'[\mathcal{R}_2]H$ and $L_1 \subseteq_{fin} (\mathsf{Loc}{-}\mathsf{dom}(h))$. By the definition of $\mathcal{R}_2$, there exists a finite location set L such that $H = \{[3{\to}5, n{\to}5] \mid n \notin L \cup \{3\}\}$. Let $H_1 = \{[3{\to}5, n{\to}5] \mid n \notin L \cup L_1 \cup \{3\}\}$. The defined set H_1 is a nonempty subset of H. We now prove that H_1 is in fact the required subset of H in the admissibility condition. Since $h'[\mathcal{R}_2]H_1$, $h'[\mathcal{R}_2]H_1$ follows from the definition of $\mathcal{R}_2$ and H_1. We also have that $\forall h_1 \in H_1.\ \mathsf{dom}(h_1) \cap L_1 = \emptyset$, because $\mathsf{dom}(h_1) \cap L_1 \subseteq \{3\}$ but L_1 does not contain 3 ($h'{=}[3{\to}1]\#L_1$).

Using the expansion operator and admissibility condition, we can define the criteria for deciding which power relation should be considered "good" for given modules (q, ϵ) and (p, η). The criteria is: a power relation should be the expansion $\mathcal{R}{\otimes}\Delta$ of an admissible $\mathcal{R}$ for the module internals (i.e., $\mathcal{R} \subseteq q \times \wp(p)$). The following lemma, which we will prove later in Sec. 5.1, provides the justification of this criteria:

> Lemma 4: For all q, p, and all power relations $\mathcal{R} \subseteq q{\times}\wp(p)$, if $\mathcal{R}$ is admissible and q is precise, then $\forall r{\in}\mathcal{F}_{noav}.\ \mathsf{prot}(r,q)[\mathsf{psim}(\mathcal{R}{\otimes}\Delta, \mathcal{R}{\otimes}\Delta)]\mathsf{prot}(r,p)$.

To see the significance of this lemma, recall that the forward simulation in Sec. 4 failed to be sound mainly because some atomic client operations are not related to themselves by fsim. The lemma indicates that as long as we are using admissible power relation $\mathcal{R}$, we do not have such a problem for psim: if $\mathcal{R}$ is admissible, then for all atomic client operations a and all environment pairs (μ', μ) with $\mathsf{psim}(\mathcal{R}{\otimes}\Delta, \mathcal{R}{\otimes}\Delta)$-related procedures, we have that $[\![a]\!]_{(q,\epsilon)}\mu'[\mathsf{psim}(\mathcal{R}{\otimes}\Delta, \mathcal{R}{\otimes}\Delta)][\![a]\!]_{(p,\eta)}\mu$.

We now define the power simulation of an abstract module (p, η) by a concrete module (q, ϵ). Let $\mathcal{R}$ be an admissible power relation such that $\mathcal{R} \subseteq q \times \wp(p)$, and let ID be the "identity" power relation defined by: $h[\mathsf{ID}]H \overset{def}{\Leftrightarrow} \{h\} = H$.

Definition 4 (Power Simulation). Module (q, ϵ) power-simulates (p, η) by $\mathcal{R}$ *iff*

$$\begin{array}{ll}
h \in p & \overset{def}{\Leftrightarrow} \exists n, n'.\, n' \neq 1 \wedge n \geq 0 \wedge h = [1{\to}n', n'{\to}n] \\
h[\eta(\mathsf{init})]v & \overset{def}{\Leftrightarrow} \text{if } (1{\notin}\mathsf{dom}(h)) \text{ then } v{=}\mathsf{flt} \text{ else } \exists n.\, n{\notin}\mathsf{dom}(h) \wedge v{=}h[1{\to}n]{\cdot}[n{\to}0] \\
h[\eta(\mathsf{inc})]v & \overset{def}{\Leftrightarrow} \text{if } (1{\notin}\mathsf{dom}(h) \vee h(1){\notin}\mathsf{dom}(h)) \text{ then } v{=}\mathsf{flt} \text{ else } v{=}h[h(1){\to}(h(h(1)){+}1)]) \\
h[\eta(\mathsf{read})]v & \overset{def}{\Leftrightarrow} \text{if } (1{\notin}\mathsf{dom}(h) \vee h(1){\notin}\mathsf{dom}(h) \vee 3{\notin}\mathsf{dom}(h)) \text{ then } v{=}\mathsf{flt} \text{ else } v{=}h[3{\to}h(h(1))] \\
h[\eta(\mathsf{final})]v & \overset{def}{\Leftrightarrow} \text{if } (1{\notin}\mathsf{dom}(h) \vee h(1){\notin}\mathsf{dom}(h)) \text{ then } v{=}\mathsf{flt} \\
& \quad\ \text{else } \exists h_0.\, v{=}h_0[1{\to}0] \wedge h{=}h_0{\cdot}[h(1){\to}h(h(1))] \\
h \in q & \overset{def}{\Leftrightarrow} \exists n.\, n \geq 0 \wedge h = [1{\to}n] \\
h[\epsilon(\mathsf{init})]v & \overset{def}{\Leftrightarrow} \text{if } (1{\notin}\mathsf{dom}(h)) \text{ then } v{=}\mathsf{flt} \text{ else } v{=}h[1{\to}0] \\
h[\epsilon(\mathsf{inc})]v & \overset{def}{\Leftrightarrow} \text{if } (1 \notin \mathsf{dom}(h)) \text{ then } v{=}\mathsf{flt} \text{ else } v{=}h[1{\to}(h(1){+}1)] \\
h[\epsilon(\mathsf{read})]v & \overset{def}{\Leftrightarrow} \text{if } (1{\notin}\mathsf{dom}(h) \vee 3{\notin}\mathsf{dom}(h)) \text{ then } v{=}\mathsf{flt} \text{ else } v{=}h[3{\to}h(1)] \\
h[\epsilon(\mathsf{final})]v & \overset{def}{\Leftrightarrow} \text{if } (1 \notin \mathsf{dom}(h)) \text{ then } v{=}\mathsf{flt} \text{ else } v{=}h[1{\to}0]
\end{array}$$

Fig. 4. Definition of Module (p, η) and (q, ϵ)

1. $\epsilon(\mathsf{init})[\mathsf{psim}(\mathsf{ID}, \mathcal{R}{\otimes}\Delta)]\eta(\mathsf{init})$ *and* $\epsilon(\mathsf{final})[\mathsf{psim}(\mathcal{R}{\otimes}\Delta, \mathsf{ID})]\eta(\mathsf{final})$*;*
2. $\forall f \in \mathsf{mop}.\ \epsilon(f)[\mathsf{psim}(\mathcal{R}{\otimes}\Delta, \mathcal{R}{\otimes}\Delta)]\eta(f)$*.*

Example 1. We demonstrate power simulation using the semantic modules (q, ϵ) and (p, η) that, respectively, correspond to counter2 and counter3 in Fig. 1. Recall that both counter2 and counter3 implement a counter "object" with two operations, inc for incrementing the counter and read for reading the value of the counter; the main difference is that counter3 uses two cells, namely cell 1 and a newly allocated one, to track the value of the counter, while counter2 uses only cell 1 for the same purpose. The corresponding semantic modules, (q, ϵ) for counter2 and (p, η) for counter3, are defined in Fig. 4. Note that the resource invariant p indicates that counter3 uses two cells 1 and n internally, and the invariant q that counter2 uses only one cell 1 internally. We will now show that the space saving in counter2 is correct, by proving that (q, ϵ) power-simulates (p, η).

The first step of power simulation is to find an admissible power relation that couples the internals of (q, ϵ) and (p, η). For this, we use the following $\mathcal{R}$:

$$h[\mathcal{R}]H \overset{def}{\Leftrightarrow} \exists L, n.\ L{\subseteq_{fin}}\mathsf{Loc} \wedge n{\geq}0 \wedge h{=}[1{\to}n] \wedge H{=}\{[1{\to}n', n'{\to}n] \mid n' {\notin} L{\cup}\{1\}\}.$$

Intuitively, $h[\mathcal{R}]H$ means that all the states in H and state h represent the same counter having the value $h(1)$, and moreover, H collectively has as many free cells as h.

The next step is to show that all the corresponding module operations of (q, ϵ) and (p, η) are related by psim. Here we only show that $\epsilon(\mathsf{init})$ and $\eta(\mathsf{init})$ are $\mathsf{psim}(\mathsf{ID}, \mathcal{R}{\otimes}\Delta)$-related. Consider h and H related by the "identity relation" ID. Then, by the definition of ID, set H must be the singleton set containing the heap h. Thus, it suffices to show that if $\mathsf{lft}(\eta(\mathsf{init}))$ does not generate an error from $\{h\}$, all the outputs of $\epsilon(\mathsf{init})$ from h are $\mathcal{R}{\otimes}\Delta$-related to some output state sets of $\mathsf{lft}(\eta(\mathsf{init}))$ from $\{h\}$. Suppose that $\mathsf{lft}(\eta(\mathsf{init}))$ does not generate an error from $\{h\}$. Then, $\eta(\mathsf{init})$ cannot output flt from h, and so, cell 1 should be in $\mathsf{dom}(h)$. From this, it follows that the concrete initialization $\epsilon(\mathsf{init})$ does not generate an error from h. We now check the non-error outputs of $\epsilon(\mathsf{init})$. When started from h, the concrete initialization $\epsilon(\mathsf{init})$ has only one non-error output, namely state $h[1{\to}0]$. We split this output state $h[1{\to}0]$ into $[1{\to}0]$ and the remainder h_0. By the definition of $\mathcal{R}$, the first part $[1{\to}0]$ of the splitting is $\mathcal{R}$-related to $H_r = \{[1{\to}n', n'{\to}0] \mid n' \notin \mathsf{dom}(h)\}$. Thus, extending $[1{\to}0]$ and H_r by the remainder h_0 gives $\mathcal{R}{\otimes}\Delta$-related state $[1{\to}0]{\cdot}h_0 = h[1{\to}0]$ and state set $H_r {*} \{h_0\}$. The state set $H_r {*} \{h_0\}$ is equal to $\{h[1{\to}n']{\cdot}[n'{\to}0] \mid n' \notin \mathsf{dom}(h)\}$, and so, it is a possible output of $\mathsf{lft}(\eta(\mathsf{init}))$ from $\{h\}$ by the definition of $\mathsf{lft}(\eta(\mathsf{init}))$. We have just shown that the output $h[1{\to}0]$ is $\mathcal{R}{\otimes}\Delta$-related to some output of $\mathsf{lft}(\eta(\mathsf{init}))$, as required.

The nondeterministic allocation in the abstract initialization $\eta(\mathsf{init})$ is crucial for the correctness of data refinement. Suppose that we change the initialization of the abstract module such that it allocates a specific cell 2:

$$h[\eta(\mathsf{init})]v \overset{def}{\Leftrightarrow} \mathsf{if}\ (1 \notin \mathsf{dom}(h))\ \mathsf{then}\ (v{=}\mathsf{flt})\mathsf{else}\ (2 \notin \mathsf{dom}(h)\ \wedge\ v{=}h[1{\to}2]{\cdot}[2{\to}0])$$

Then, (q, ϵ) no longer data-refines (p, η);[9] by testing the allocation status of cell 2 using memory allocation and pointer comparison, a client command can detect the replacement of (p, η) by (q, ϵ), and exhibit a behavior that is only possible with (q, ϵ), but not with (p, η). Power simulation correctly captures this failure of data refinement. More specifically, for all power relations $\mathcal{R} \subseteq q \times \wp(p)$ if $\epsilon(\mathsf{init})[\mathsf{psim}(\mathsf{ID}, \mathcal{R}{\otimes}\Delta)]\eta(\mathsf{init})$, then $\mathcal{R}$ cannot be admissible. To see the reason, suppose that $\epsilon(\mathsf{init})[\mathsf{psim}(\mathsf{ID}, \mathcal{R}{\otimes}\Delta)]\eta(\mathsf{init})$. When $\epsilon(\mathsf{init})$ and $\mathsf{lft}(\eta(\mathsf{init}))$ are run from ID-related $[1{\to}0]$ and $\{[1{\to}0]\}$, $\epsilon(\mathsf{init})$ outputs $[1{\to}0]$ and $\mathsf{lft}(\eta(\mathsf{init}))$ outputs $\{[1{\to}2, 2{\to}0]\}$ or $\emptyset$. Thus, by the definition of $\mathsf{psim}(\mathsf{ID}, \mathcal{R}{\otimes}\Delta)$, $[1{\to}0]$ should be $\mathcal{R}{\otimes}\Delta$-related to $\{[1{\to}2, 2{\to}0]\}$ or $\emptyset$. Then, by the definition of $\mathcal{R}{\otimes}\Delta$, state $[1{\to}0]$ is $\mathcal{R}$-related to $\{[1{\to}2, 2{\to}0]\}$ or $\emptyset$. In either case, $\mathcal{R}$ is not admissible; the first case violates the second conjunct about the free cells in the admissibility condition, and the second case violates the first conjunct about the nonemptiness. □

5.1 Soundness of Power Simulation

The soundness of power simulation follows from the fact that every atomic client operation is related to itself by psim (Lemma 4), all language constructs preserve psim (Lemma 5,6) and $\mathsf{psim}(\mathsf{ID}, \mathsf{ID})$ is precisely the improvement requirement in the definition of data refinement (Lemma 7). Among these lemmas, we give the proof of only the most important one, Lemma 4. Then, we show how all the lemmas are used to give the soundness of power simulation. The missing proofs appear in the full version of this paper [13].

Lemma 4. *For all predicates q, p, and all power relations $\mathcal{R} \subseteq q \times \wp(p)$, if q is precise and $\mathcal{R}$ is admissible, then $\forall r{\in}\mathcal{F}_{noav}.\ \mathsf{prot}(r, q)[\mathsf{psim}(\mathcal{R}{\otimes}\Delta, \mathcal{R}{\otimes}\Delta)]\mathsf{prot}(r, p)$.*

[9] Even when we replace p by a more precise invariant $\{[1{\to}2, 2{\to}n] \mid n \geq 0\}$, module (q, ϵ) does not data-refine (p, η).

Note that the lemma requires that the "resource invariant" q for the "concrete module" be precise, and that r be a av-free finite local action. None of these requirements can be omitted, because the requirements are used crucially in the proof of the lemma.

Proof. Let r_q be $\mathsf{prot}(r,q)$ and let r_p be $\mathsf{prot}(r,p)$. Pick arbitrary $[\mathcal{R}\otimes\Delta]$-related h and H such that $\mathsf{lft}(r_p)$ does not generate an error from H. Since $h[\mathcal{R}\otimes\Delta]H$, state h and state set H can, respectively, be split into $h_q{\cdot}h_0 = h$ and $H = H_p * \{h_0\}$ for some h_q, h_0, H_p such that $h_q[\mathcal{R}]H_p$. We note two facts about these splittings. First, set H_p contains a state that is disjoint from h_0. Since h_q and H_p are related by the admissible relation $\mathcal{R}$ and $\mathsf{dom}(h_q)$ is disjoint from $\mathsf{dom}(h_0)$, there is a nonempty subset of H_p such that every h_1 in the subset satisfies $h_1\#\mathsf{dom}(h_0)$. We pick one state from this subset, and call it h_p. Second, the state h_p in H_p and the part h_q of the splitting of h, respectively, belong to p and q. This second fact follows since $h_q[\mathcal{R}]H_p$ and $\mathcal{R} \subseteq q \times \wp(p)$. We sum up the obtained properties about H_p, h_0, h_q, h_p below:

$$H = H_p * \{h_0\} \;\wedge\; h = h_q{\cdot}h_0 \;\wedge\; h_q[\mathcal{R}]H_p \;\wedge\; h_p\#h_0 \;\wedge\; h_p \in p \;\wedge\; h_q \in q.$$

We now prove that r_q does not generate an error from h. Since the lifted command $\mathsf{lft}(r_p)$ does not generate an error from H and state $h_p{\cdot}h_0$ is in this input state set H, we have that $\neg h_p{\cdot}h_0[r_p]\mathsf{flt} \;\wedge\; \neg h_p{\cdot}h_0[r_p]\mathsf{av}$. This absence of errors of r_p ensures one important property of r: r cannot generate flt from h_0. To see the reason, note that h_p is in p, and that $\neg h_p{\cdot}h_0[r]\mathsf{flt}$ since $\neg h_p{\cdot}h_0[r_p]\mathsf{flt}$. So, if $h_0[r]\mathsf{flt}$, then by the definition of prot, we have that $h_p{\cdot}h_0[r_p]\mathsf{av}$, which contradicts $\neg h_p{\cdot}h_0[r_p]\mathsf{av}$. We will use this property of r to show $\neg h[r_q]\mathsf{flt}$ and $\neg h[r_q]\mathsf{av}$. Since $h = h_0{\cdot}h_q$ and $\neg h_0[r]\mathsf{flt}$, by the safety monotonicity of r, we have that $\neg h[r]\mathsf{flt}$. Thus, $\neg h[r_q]\mathsf{flt}$ by the definition of prot. For $\neg h[r_q]\mathsf{av}$, we have to show that

$$\neg h[r]\mathsf{av} \wedge \big(h[r]\mathsf{flt} \vee \big(\forall m_q, m_0 \in \mathsf{St}.\ (m_q{\cdot}m_0{=}h \wedge m_q \in q) \Rightarrow \neg m_0[r]\mathsf{flt}\big)\big).$$

Since r is av-free, it does not output av for any input states. For the second conjunct, consider a splitting $m_q{\cdot}m_0$ of h such that $m_q \in q$. Then, since $h = h_q{\cdot}h_0$, $h_q \in q$ and q is precise, we should have that $m_q = h_q$ and $m_0 = h_0$. Since $\neg h_0[r]\mathsf{flt}$, it follows that $\neg m_0[r]\mathsf{flt}$.

Finally, we prove that every output state of r_q from h is $\mathcal{R}\otimes\Delta$-related to some output state set of $\mathsf{lft}(r_p)$ from H. In the proof, we will use $\neg h_0[r]\mathsf{flt}$, which we have shown in the previous paragraph. Consider a state h' such that $h[r_q]h'$. Since $h = h_0{\cdot}h_q$, by the definition of $\mathsf{prot}(r,q)$, we have that $h_0{\cdot}h_q[r]h'$. Since $\neg h_0[r]\mathsf{flt}$, we can apply the frame property of r to this computation, and obtain a substate h'_0 of h' such that $h' = h'_0{\cdot}h_q$. Let L_0 be the finite set that includes all the indirectly accessed locations by the "computation" $h_0{\cdot}h_q[r]h'_0{\cdot}h_q$; L_0 is guaranteed to exist by the finite access property of r. Let L be the location set $\big(L_0 \cup \mathsf{dom}(h_0) \cup \mathsf{dom}(h'_0)\big) - \mathsf{dom}(h_q)$. Since $h_q[\mathcal{R}]H_p$ and $\mathcal{R}$ is admissible, there is a subset H_1 of H_p such that

$$H_1 \subseteq H_p \;\wedge\; H_1[\mathcal{R}]h_q \;\wedge\; \forall h_1 \in H_1.\ h_1\#L.$$

We will show that $H_1 * \{h_0'\}$ is the required output state set. Since h_q and H_1 are $\mathcal{R}$-related, their h_0'-extensions, $h_q{\cdot}h_0'$ and $H_1 * \{h_0'\}$, have to be $\mathcal{R}{\otimes}\varDelta$-related. Thus, it remains to show that $H = H_p * \{h_0\}[\mathsf{lft}(r_p)]H_1 * \{h_0'\}$. Instead of proving this relationship directly, we will prove that

$$H_1 * \{h_0\}[\mathsf{lft}(r_p)]H_1 * \{h_0'\}.$$

Because, then, the definition of $\mathsf{lft}(r_p)$ will ensure that we also have the required computation. For every m in $H_1 * \{h_0'\}$, there is a state $m_1 \in H_1$ such that $m = m_1{\cdot}h_0'$. By the choice of H_1, we have $m_1 \in H_p \wedge m_1\#L$. Then, there exist splitting $n_1{\cdot}n_2 = m_1$ of m_1 and splitting $o_2{\cdot}o_3 = h_q$ of h_q with the property that $n_1\#h_q$ and $\mathsf{dom}(n_2) = \mathsf{dom}(o_2)$. Note that $n_1\#h_q$ implies $n_1\#L_0$, because $L_0 \subseteq L \cup \mathsf{dom}(h_q)$ and $n_1{\cdot}n_2\#L$. We obtain a new computation of r_p as follows:

$$\begin{array}{lll}
h_q{\cdot}h_0[r]h_q{\cdot}h_0' & \Longrightarrow n_1{\cdot}h_q{\cdot}h_0[r]n_1{\cdot}h_q{\cdot}h_0' & (\because \text{the finite access property of } r)\\
& \Longrightarrow n_1{\cdot}o_2{\cdot}o_3{\cdot}h_0[r]n_1{\cdot}o_2{\cdot}o_3{\cdot}h_0' & (\because h_q = o_2{\cdot}o_3)\\
& \Longrightarrow n_1{\cdot}n_2{\cdot}o_3{\cdot}h_0[r]n_1{\cdot}n_2{\cdot}o_3{\cdot}h_0' & (\because \text{the contents independence of } r)\\
& \Longrightarrow n_1{\cdot}n_2{\cdot}h_0[r]n_1{\cdot}n_2{\cdot}h_0' & (\because \text{the frame property of } r)\\
& \Longrightarrow m_1{\cdot}h_0[r]m & (\because m_1 = n_1{\cdot}n_2 \wedge m_1{\cdot}h_0' = m)\\
& \Longrightarrow m_1{\cdot}h_0[r_p]m & (\because \text{the definition of } \mathsf{prot}(r,p))
\end{array}$$

Note that the input $m_1{\cdot}h_0$ of the obtained computation belongs to the state set $H_1{*}\{h_0\}$. We just have shown $H_1{*}\{h_0\}[\mathsf{lft}(r_p)]H_1{*}\{h_0'\}$. □

Lemma 5. *For all power relations $\mathcal{R}_0, \mathcal{R}_1, \mathcal{R}_2$ and all FLAs r_0, r_0', r_1, r_1', if $r_0'[\mathsf{psim}(\mathcal{R}_0,\mathcal{R}_1)]r_0 \wedge r_1'[\mathsf{psim}(\mathcal{R}_1,\mathcal{R}_2)]r_1$, then $\mathsf{seq}(r_0', r_1')[\mathsf{psim}(\mathcal{R}_0,\mathcal{R}_2)]\mathsf{seq}(r_0, r_1)$.*

Lemma 6. *For all power relations $\mathcal{R}_0, \mathcal{R}_1$, sets I and I-indexed families $\{r_i'\}_{i\in I}$, $\{r_i\}_{i\in I}$ of FLAs, if $\forall i{\in}I.r_i'[\mathsf{psim}(\mathcal{R}_0,\mathcal{R}_1)]r_i$, then $\bigcup_{i\in I} r_i'[\mathsf{psim}(\mathcal{R}_0,\mathcal{R}_1)] \bigcup_{i\in I} r_i$.*

Theorem 1 (Abstraction). *Let $(q,\epsilon), (p,\eta)$ be semantic modules, and $\mathcal{R}$ be an admissible power relation s.t. $\mathcal{R} \subseteq q \times \wp(p)$. If (q,ϵ) power-simulates (p,η) by $\mathcal{R}$, then for all commands C and all environments μ, μ', we have that*

$$(\forall P.\ \mu'(P)[\mathsf{psim}(\mathcal{R}{\otimes}\varDelta, \mathcal{R}{\otimes}\varDelta)]\mu(P)) \Rightarrow [\![C]\!]_{(q,\epsilon)}\mu'[\mathsf{psim}(\mathcal{R}{\otimes}\varDelta, \mathcal{R}{\otimes}\varDelta)][\![C]\!]_{(p,\eta)}\mu.$$

Proof. We use induction on the structure of C. When C is a module operation f or a procedure name P, the theorem follows from the assumption: (q,ϵ) power-simulates (p,η) by $\mathcal{R}$, and for all P, $\mu'(P)[\mathsf{psim}(\mathcal{R}{\otimes}\varDelta, \mathcal{R}{\otimes}\varDelta)]\mu(P)$. When C is an atomic client operation a, the theorem holds because of Lemma 4. The remaining three cases follow from the closedness of $\mathsf{psim}(\mathcal{R}{\otimes}\varDelta, \mathcal{R}{\otimes}\varDelta)$ in Lemma 5 and 6: $\mathsf{psim}(\mathcal{R}{\otimes}\varDelta, \mathcal{R}{\otimes}\varDelta)$ is closed under arbitrary union and seq. This closedness property directly implies that the induction step goes through for the cases of $C_1[]C_2$ and $C_1;C_2$. For $\mathsf{fix}\,P.C'$, we note that the closedness under arbitrary union implies that $\mathsf{psim}(\mathcal{R}{\otimes}\varDelta, \mathcal{R}{\otimes}\varDelta)$ is complete,[10] and that this completeness is what we need to prove the induction step for $\mathsf{fix}\,P.C'$. □

[10] $\mathsf{psim}(\mathcal{R}_1, \mathcal{R}_2)$ relates the least FLA to itself, and is chain-complete.

Lemma 7 (Identity Extension). *A module* (q, ϵ) *data-refines another module* (p, η) *iff for all complete commands* C*, we have that* $[\![C]\!]^c_{(q,\epsilon)}[\mathsf{psim}(\mathsf{ID}, \mathsf{ID})][\![C]\!]^c_{(p,\eta)}$.

Theorem 2 (Soundness). *If a module* (q, ϵ) *power-simulates another module* (p, η) *by an admissible power relation* $\mathcal{R} \subseteq q \times \wp(p)$*, then* (q, ϵ) *data-refines* (p, η).

Proof. Suppose that a module (q, ϵ) power-simulates another module (p, η) by an admissible power relation $\mathcal{R} \subseteq q \times \wp(p)$. We will show that for all complete commands C, $[\![C]\!]_{(q,\epsilon)}[\mathsf{psim}(\mathsf{ID}, \mathsf{ID})][\![C]\!]_{(p,\eta)}$, because, then, module (q, ϵ) should data-refine (p, η) (Lemma 7). Pick an arbitrary complete program C. Let μ be an environment that maps all program identifiers to the empty relation. By Lemma 6, $\mu(P)[\mathsf{psim}(\mathcal{R}\otimes\Delta, \mathcal{R}\otimes\Delta)]\mu(P)$ for all P in pid. From this, we derive the required relationship as follows:

$$\begin{array}{ll} (\forall P \in \mathsf{pid}.\ \mu(P)[\mathsf{psim}(\mathcal{R}\otimes\Delta, \mathcal{R}\otimes\Delta)]\mu(P)) & \\ \Longrightarrow\ [\![C]\!]_{(q,\epsilon)}\mu[\mathsf{psim}(\mathcal{R}\otimes\Delta, \mathcal{R}\otimes\Delta)][\![C]\!]_{(p,\eta)}\mu & (\because \text{Theorem 1}) \\ \Longrightarrow\ [\![C]\!]^c_{(q,\epsilon)}[\mathsf{psim}(\mathsf{ID}, \mathsf{ID})][\![C]\!]^c_{(p,\eta)} & (\because \text{Lemma 5 and Def. of } [\![-]\!]^c) \end{array}$$

□

6 Conclusion

In this paper, we have proposed a new data-refinement method, called power simulation, for programs with low-level pointer operations, and provided a non-trivial soundness proof of the method.

The very idea of relating a state to a state set in power simulation comes from Reddy's method for data refinement [16]. In order to have a single complete data-refinement method for a language *without pointers*, he lifted forward simulation such that all the components of the simulation become about state sets, instead of states. However, the details of the two methods, such as the admissibility condition for coupling relations and the lifting operator for commands, are completely different.

References

1. A. Banerjee and D. A. Naumann. Representation independence, confinement and access control (extended abstract). In *POPL'02*, pages 166–177. ACM, 2002.
2. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL'03*, pages 213–223. ACM, 2003.
3. D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *ECOOP'01*, volume 2072 of *Lecture Notes in Computer Science*, pages 53–76. Springer-Verlag, 2001.
4. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1998.
5. E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976.
6. J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *ESOP'86*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer-Verlag, 1986.

7. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
8. J. Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPLA'91*, pages 271–285. ACM, 1991.
9. J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.
10. S. Istiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL'01*, pages 14–26, London, 2001. ACM.
11. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Trans. on Program. Lang. and Syst.*, 24(5):491–553, 2002.
12. I. Mijajlovic, N. Torp-Smith, and P. O'Hearn. Refinement and separation contexts. In *FSTTCS'04*, volume 3328 of *Lecture Notes in Computer Science*, pages 421–433. Springer-Verlag, 2004.
13. I. Mijajlovic and H. Yang. Data refinement with low-level pointer operations. Manuscript, 2005. Available at http://ropas.snu.ac.kr/˜ hyang/paper/full-ps.ps.
14. D. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. In *LICS'04*, pages 313–323. IEEE, 2004.
15. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL'04*, pages 268–280, Venice, 2004. ACM.
16. U. S. Reddy. Talk at MFPS'00, Hokoken, New Jersey, USA, 2000.
17. U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1):257–305, 2004.
18. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, volume 17, pages 55 – 74, Copenhagen, 2002. IEEE.
19. I. Stark. Categorical models for local names. *Lisp and Symbolic Comput.*, 9(1):77–107, 1996.

A Simple Semantics for Polymorphic Recursion[⋆]

William L. Harrison

Dept. of Computer Science, University of Missouri,
Columbia, Missouri, USA

Abstract. Polymorphic recursion is a useful extension of Hindley-Milner typing and has been incorporated in the functional programming language Haskell. It allows the expression of efficient algorithms that take advantage of non-uniform data structures and provides key support for generic programming. However, polymorphic recursion is, perhaps, not as broadly understood as it could be and this, in part, motivates the denotational semantics presented here. The semantics reported here also contributes an essential building block to any semantics of Haskell: a model for first-order polymorphic recursion. Furthermore, Haskell-style type classes may be described within this semantic framework in a straightforward and intuitively appealing manner.

1 Introduction

This paper presents a denotational semantics for an extension to the Hindley-Milner type system [18] called *polymorphic recursion* [21,16,8,1]. Polymorphic recursion (sometimes called *non-uniform* recursion) allows functions in which the type of a recursive call differs from that of the function itself. The approach taken here conservatively extends the semantics for Hindley-Milner polymorphism due to Ohori [22,23]. Ohori's semantics—the *simple model of ML polymorphism*—proceeds by construction from any frame model [20] of the simply-typed λ-calculus. The principal technical result of this work is that any such Ohori model may be extended to a model of first-order polymorphic recursion in a straightforward and natural manner. A compelling prospect for this approach is as a model for ad hoc polymorphism [29,13,15]. We describe how the simple model of polymorphic recursion provides a foundation for Haskell-style type classes and how the approach relates to a previous denotational model [28].

Polymorphic recursion is part of the functional language Haskell [25] and is a building block for generic programming [11,10] as well. Its presence in Haskell allows the straightforward expression of efficient data structures and algorithms; consider this example (adapted from Okasaki [24], page 142):

$$\begin{array}{ll}
\textbf{data}\ \mathit{Seq}\ a\ =\ \mathit{Nil} \mid \mathit{SCons}\ a\ (\mathit{Seq}\,(a,a)) & \mathit{stl}\ ::\ \mathit{Seq}\ a \rightarrow \mathit{Seq}\,(a,a) \\
\mathit{size}\ \ ::\ \mathit{Seq}\ a \rightarrow\ \mathit{Int} & \mathit{stl}\ (\mathit{SCons}\ \_\ t)\ = t \\
\mathit{size}\ s\ = \textbf{if}\ \mathit{isNil}\ s\ \textbf{then}\ 0\ \textbf{else}\ 1 + 2 * \mathit{size}\ (\mathit{stl}\ s) &
\end{array}$$

⋆ This research supported in part by subcontract GPACS0016, System Information Assurance II, through OGI/Oregon Health & Sciences University.

The *Seq* data type represents sequences compactly so as to render their counting more efficiently than in a standard list representation. The *size* function calculates the length of a sequence and runs in $O(log\ n)$ time while its list analogue, *length*, runs in $O(n)$ time. Here, *Seq* only encodes sequences of length $2^n - 1$ for some n, although it is simple to extend its definition to capture sequences of arbitrary length by including an "even" constructor of type $\mathit{ECons} :: (\mathit{Seq}\,(a,a)) \to \mathit{Seq}\ a$; see, for example, Okasaki [24], page 144, for further details. Note that the recursive call to *size* occurs at an *instance* of its declared type: $\mathit{Seq}\,(a,a) \to \mathit{Int}$. Haskell requires explicit type signatures for polymorphic recursive definitions as type inference is undecidable in the presence of polymorphic recursion [8].

It has been recognized since its inception that the Hindley-Milner type system is more restrictive than is desirable as the following example (due to Milner [18]) makes clear. Consider the following standard ML definitions:

$$\mathbf{fun}\, f\ x\ =\ x;\ \ \mathbf{fun}\, g\ y\ =\ f\ 3; \qquad\qquad \mathbf{fun}\, f\ x\ =\ x\ \ \mathbf{and}\, g\ y\ =\ f\ 3;$$

When f is defined independently of g (left), it has type $\forall a.a \to a$ as one might expect, but when defined mutually (right), it has, unexpectedly, the less general type $\mathit{int} \to \mathit{int}$. Polymorphic recursion was initially developed [21] to overcome such "anomalies" in Hindley-Milner typing. Polymorphic recursive type systems maintain the type signatures of recursive definitions in universally quantified form to avoid such unintended typings. More will be said about this in Section 2.

Consider the polymorphic (but not polymorphic recursive) function *length*:

$$\begin{array}{l} \mathit{length} :: [a] \to \mathit{Int} \\ \mathit{length} = \lambda\, x.\ \mathbf{if}\ \mathit{null}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ 1{+}(\mathit{length}\ (\mathit{tail}\ x)) \end{array}$$

An Ohori-style semantics denotes *length* by the collection of its meanings at ground instances: $\{\ \langle \tau,\ \mathit{len}_\tau \rangle \mid \tau = [\mathit{Int}] \to \mathit{Int}, \ldots \}$ where each len_τ is defined:

$$\mathit{len}_\tau = \mathit{fix}\,(\lambda l.\ [\![\lambda x.\ \mathbf{if}\ \mathit{null}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ 1{+}(\mathit{length}\ (\mathit{tail}\ x))]\!]\rho[\mathit{length} \mapsto l])$$

Here, *fix* is the least fixed point operator defined conventionally on a domain D_τ denoting τ. One is tempted to try a similar definition for the polymorphic recursive function *size*:

$$\mathit{size}_\tau = \mathit{fix}\ (\lambda s.\ [\![\lambda x.\ \mathbf{if}\ \mathit{isNil}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ 1 + 2 * \mathit{size}\ (\mathit{stl}\ x))]\!]\rho[\mathit{size} \mapsto s]) \qquad (\dagger)$$

But, where does this *fix* "live"? Or, in other words, over which domain is it defined? Intuitively, here's the problem: if the "input *size*" (i.e., the "s" in "$\mathit{fix}\ (\lambda s \ldots$") lives in D_τ for $\tau = \mathit{Seq}(\tau') \to \mathit{Int}$, then the "output *size*" (i.e., the "*size*" in "$\mathit{size}(\mathit{stl}\ x)$") lives in the domain for $\tau = \mathit{Seq}(\tau', \tau') \to \mathit{Int}$. The above definition does not appear to make sense in any fixed D_τ as was the case with *length*. We answer this question precisely in Section 3.2 below.

The Girard-Reynolds calculus (also known as *System F* [2] and the *polymorphic λ-calculus* [26]) allows lambda abstraction and application over types as well as over values; as such, it is sometimes referred to as a second-order

λ-calculus. Denotational models of second-order λ-calculi exist (e.g., the PER model described in [2]). Such models provide one technique for specifying ML polymorphism[1]. Harper and Mitchell take this approach [5,19] for the core of Standard ML called core-ML. They translate a core-ML term (i.e., one without type abstraction or application) into a second-order core-XML term (i.e., one with type abstraction and application). A core-ML term is then modeled by the denotation of its translation in an appropriate model of core-XML.

ML polymorphism is considerably more restrictive than that of a second-order λ-calculus; type quantification and lambda abstraction occur only over base types and values. Because of its restrictiveness relative to the Girard-Reynolds calculus, it is possible to give a predicative semantics to ML polymorphism [23,22] that is a conservative extension of the frame semantics of the simply-typed λ-calculus. Ohori's model of ML polymorphism is appealing precisely because of its simplicity. It explains ML polymorphism in terms of simpler, less expressive things (such as the frame semantics of the simply-typed λ-calculus) rather than in terms of inherently richer and more expressive things (such as the semantics of the second-order λ-calculus). The model of polymorphic recursion presented here retains this simplicity.

The rest of this paper proceeds as follows. Section 2 presents the language for first-order polymorphic recursion; the semantics of this language is then formulated in Section 3. Section 3 begins with an overview of frame semantics and Section 3.1 presents the simple model of ML polymorphism. Sections 3.2 and 3.3 demonstrate how a straightforward extension of Ohori's model provides a semantic foundation for first-order polymorphic recursion. Section 4 motivates the application of this semantic framework to Haskell-style ad hoc polymorphism. Section 5 discusses the rôle of the present research in the semantics of the Haskell language and outlines future directions. Related work is discussed throughout rather than in a separate section.

2 A Language and Type System for Polymorphic Recursion

The language we consider—called PR hereafter—is shown in Figure 1; it is the first-order type system referred to as ML/1′ by Kfoury, et al. [16]. Kfoury, et al. describe several type systems of increasing expressiveness that extend Hindley-Milner with polymorphic recursion. PR is "syntax-oriented," meaning that derivations in this type system are unique (modulo the order of application of the type generalization rule GEN); this has a useful (albeit not strictly necessary) virtue w.r.t. the coherence of the semantic equations.

The type language for PR (following Kfoury, et al. [16]) is stratified into "open" and "universal" types ($\mathbf{T}_0$ and $\mathbf{T}_1$, respectively) and, following Ohori

[1] Following Ohori [23,22], we shall refer to the first-order variety of polymorphism occurring in Haskell and ML as *ML polymorphism* as both languages use varieties of Hindley-Milner polymorphism [9,18].

simple types	$\tau \in \mathsf{Type}$	$\tau ::= b \mid (\tau \rightarrow \tau')$
open types	$\gamma \in \mathbf{T}_0$	$\gamma ::= \alpha \mid (\gamma \rightarrow \gamma') \mid \tau$
universal types	$\sigma \in \mathbf{T}_1$	$\sigma ::= \forall\, \alpha.\, \sigma \mid \gamma$
expressions		$M ::= x \mid (M\ N) \mid (\lambda\, x.\, M) \mid \text{let } x = N \text{ in } M \mid \text{pfix } x.M$

$$\text{GEN}\ \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall\alpha.\sigma}\ {\scriptstyle (\alpha \notin FV(\Gamma))} \qquad \text{APP}\ \frac{\Gamma \vdash M : \gamma \rightarrow \gamma' \quad \Gamma \vdash N : \gamma}{\Gamma \vdash (M\ N) : \gamma'}$$

$$\text{ABS}\ \frac{\Gamma, x : \gamma \vdash M : \gamma'}{\Gamma \vdash (\lambda x.M) : \gamma \rightarrow \gamma'} \qquad \text{LET}\ \frac{\Gamma \vdash N : \sigma' \quad \Gamma, x : \sigma' \vdash M : \sigma}{\Gamma \vdash (\text{let } x = N \text{ in } M) : \sigma}$$

$$\text{VAR}\ \frac{\Gamma(x) = \sigma, \sigma \preceq_s \gamma}{\Gamma \vdash x : \gamma} \qquad \text{PFIX}\ \frac{\Gamma, x : \forall\bar{\alpha}.\gamma' \vdash M : \gamma' \quad \forall\bar{\alpha}.\gamma' \preceq \gamma}{\Gamma \vdash \text{pfix } x.M : \gamma}\ {\scriptstyle (\bar{\alpha} \notin FV(\Gamma))}$$

Fig. 1. The type system and expression syntax of PR. The type language of PR is stratified into *simple*, *open*, and *universal* types. PR departs from Hindley-Milner in its VAR and PFIX rules and polymorphic recursion is manifested in the PFIX rule.

[23,22], includes the set of "simple" (i.e., ground) types Type. Variables τ, γ, and σ refer (whether subscripted or not) to members of Type, $\mathbf{T}_0$, and $\mathbf{T}_1$, respectively, throughout this article. The abstract syntax for PR includes an explicit fix-point operator for polymorphic recursion called "pfix".

We will sometimes write a universal type $\sigma = \forall\alpha_1.\ldots\forall\alpha_n.\gamma$ for $0 \leq n$ as $\forall\bar{\alpha}.\gamma$ where $\bar{\alpha}$ is a (possibly) empty set of type variables. The relation $\sigma \preceq \gamma$ holds when γ is an instance of σ. Formally, open type γ is an *instantiation* of $\forall\bar{\alpha}.\gamma'$ for substitution s (written $(\forall\bar{\alpha}.\gamma') \preceq_s \gamma$), if, and only if, for some open types $\gamma_1 \cdots \gamma_n$ in $\mathbf{T}_0$, $s = [\alpha_1 \mapsto \gamma_1, \ldots, \alpha_n \mapsto \gamma_n]$ and $\gamma = s\gamma'$.

The rules GEN, APP, and ABS are standard and require no further comment. The LET rule captures let-polymorphism in the usual way, although it has a slightly different form than one might expect. Both the let term in the conclusion of the rule and the M term in its hypothesis have universal type σ where one would typically find an open type γ. This is inherited from ML/1$'$ and its effect is merely superficial [16]; one could change the universal σ to an open γ without affecting the language defined by the type system.

Polymorphic recursion arises in PR because, in the antecedent of the PFIX rule, the type binding for the recursively defined variable, $x : \forall\bar{\alpha}.\gamma'$, contains quantifiers if $\bar{\alpha} \neq \emptyset$. Consequently, x can be applied at any instance of its type (i.e., at any γ_i such that $\forall\bar{\alpha}.\gamma' \preceq \gamma_i$). The rule allowing such instantiations is VAR. VAR performs both variable look-up and instantiation—rôles that would typically be performed by two separate rules. VAR looks up a variable binding, $x : \sigma$, in the type environment Γ and its conclusion types the variable at an instantiation of that binding, $x : \gamma$.

Figure 2 presents the outline of a type derivation for *size* in PR illustrating how PR accommodates polymorphic recursion. Assume that Γ_0 contains bindings

$$\dfrac{\dfrac{\dfrac{(\ddagger)}{\Gamma_1 \vdash \lambda s.\textbf{if } \mathit{isNil}\, s \textbf{ then } 0 \textbf{ else } 1+2*\mathit{size}(\mathit{stl}\ s) : \mathit{Seq}(\alpha)\to \mathit{Int}}\ \text{ABS}}{\Gamma_0 \vdash \text{pfix } \mathit{size}.\lambda s.\textbf{if } \mathit{isNil}\, s \textbf{ then } 0 \textbf{ else } 1+2*\mathit{size}(\mathit{stl}\ s) : \mathit{Seq}(\alpha)\to \mathit{Int}}\ \text{PFIX}(\alpha)}{\Gamma_0 \vdash \text{pfix } \mathit{size}.\lambda s.\textbf{if } \mathit{isNil}\, s \textbf{ then } 0 \textbf{ else } 1+2*\mathit{size}(\mathit{stl}\ s) : \forall\alpha.\mathit{Seq}(\alpha)\to \mathit{Int}}\ \text{GEN}(\alpha)$$

$$\dfrac{\dfrac{\begin{array}{l}\Gamma_2(\mathit{size})=\forall\alpha.(\mathit{Seq}\,\alpha)\to \mathit{Int}\\ (\forall\alpha.(\mathit{Seq}\,\alpha)\to \mathit{Int})\preceq_s \mathit{Seq}(\alpha\times\alpha)\to \mathit{Int}\\ s=[\alpha\mapsto\alpha\times\alpha]\end{array}}{\Gamma_2\vdash \mathit{size}: \mathit{Seq}(\alpha\times\alpha)\to \mathit{Int}}\ \text{VAR} \qquad \dfrac{\dfrac{}{\ldots}}{\Gamma_2 \vdash (\mathit{stl}\ s): \mathit{Seq}\,(\alpha\times\alpha)}}{\Gamma_2\vdash \mathit{size}\ (\mathit{stl}\ s): \mathit{Int}}$$

Fig. 2. Type Checking *size* in PR. The type derivation for *size* in PR appears in the upper derivation. The second derivation (below) occurs within the "(‡)" above. Certain routine parts are suppressed for the sake of readability.

for arithmetic functions, *stl*, *isNil*, *SCons* and *Nil*, etc., Γ_1 extends Γ_0 with the binding $\mathit{size} : \forall\alpha.\,(\mathit{Seq}\,\alpha)\to\ \mathit{Int}$, and Γ_2 extends Γ_1 with $s : (\mathit{Seq}\,\alpha)$. It is at the leaves of the derivation in the bottom half of Figure 2 that this derivation becomes interesting; the application of the VAR rule instantiates *size*, not at its defined type, $\mathit{Seq}(\alpha)\to \mathit{Int}$, but rather at the instance $\mathit{Seq}(\alpha\times\alpha)\to\mathit{Int}$. This is precisely where polymorphic recursion manifests itself within *size*.

3 The Simple Model of Polymorphic Recursion

This section presents the semantics of PR. First, we briefly overview the frame semantics of the simply-typed λ-calculus and then, in Section 3.1, outline how this semantics is extended in the simple model of ML polymorphism. Section 3.2 describes the semantic setting where polymorphic recursive fixed-point equations are solved; Section 3.3 describes the semantics of PR in this setting.

Background on Type-frames Semantics. One may think of a frame model as set-theoretic version of a cartesian closed category. That is, it provides "objects" (i.e., D_τ for each simple type τ) and axioms of representability and extensionality characterizing functions from objects to objects in terms of an application operator $\bullet$. In this article, each simple type model D_τ is presumed to be built from sets with additional structure and we write $|D_\tau|$ for the underlying set of D_τ. We refer to D_τ as a *frame object* and to $|D_\tau|$ as its *frame set*. We assume that each $|D_\tau| \neq \emptyset$. A *frame* is a pair $\langle \mathcal{D}, \bullet\rangle$ where

1. $\mathcal{D} = \{D_\tau \mid \tau \in \mathsf{Type}\ \}$ and $\mathcal{D}\neq\emptyset$
2. $\bullet$ is a family of operations $\bullet_{\tau_1\tau_2} \in |D_{(\tau_1\to\tau_2)}|\to|D_{\tau_1}|\to|D_{\tau_2}|$

The set function $\phi : |D_{\tau_1}|\to|D_{\tau_2}|$ is *representable* in $|D_{(\tau_1\to\tau_2)}|$ if

$$\exists\, f\in|D_{(\tau_1\to\tau_2)}| \text{ s.t. } \phi(d) = f\bullet_{\tau_1\tau_2} d,\quad \forall d\in|D_{\tau_1}|$$

The frame $\langle \mathcal{D}, \bullet \rangle$ is *extensional* if, for all $f, g \in |D_{(\tau_1 \to \tau_2)}|$,

$$\text{if } f \bullet_{\tau_1 \tau_2} d = g \bullet_{\tau_1 \tau_2} d \text{ for all } d \in |D_{\tau_1}|, \text{ then } f = g$$

A value environment ρ is *compatible* with a type environment $\mathcal{A}$ when

$$\mathsf{dom}(\rho) = \mathsf{dom}(\mathcal{A}), \text{ and } \rho\, x \in |D_\tau| \text{ iff } (x : \tau) \in \mathcal{A}$$

The compatibility relation is designated by $\mathcal{A} \models \rho$. The set of value environments compatible with $\mathcal{A}$ is designated Env($\mathcal{A}$). Let $\langle \mathcal{D}, \bullet \rangle$ be any frame, $\lambda^{\to}$ be the simply-typed λ-calculus and ρ a value environment such that $\mathcal{A} \models \rho$. Then, the map $\mathcal{D}[\![-]\!] \in \lambda^{\to} \to Env \to (\bigcup |D_\tau|)$ obeys the *environment model condition* if the following equations hold:

$$\begin{array}{ll}
\mathcal{D}[\![\mathcal{A} \vdash x : \tau]\!]\rho & = \rho\ x \\
\mathcal{D}[\![\mathcal{A} \vdash (M\ N) : \tau]\!]\rho & = (\mathcal{D}[\![\mathcal{A} \vdash M : \tau' \to \tau]\!]\rho) \bullet (\mathcal{D}[\![\mathcal{A} \vdash N : \tau']\!]\rho) \\
\mathcal{D}[\![\mathcal{A} \vdash \lambda x : \tau_1 . M : \tau_1 \to \tau_2]\!]\rho & = \text{ the } f \in |D_{\tau_1 \to \tau_2}| \text{ s.t. for all } d \in |D_{\tau_1}|, \\
& \qquad f \bullet d = \mathcal{D}[\![\mathcal{A}, x : \tau_1 \vdash M : \tau_2]\!]\rho[x \mapsto d]
\end{array}$$

For any extensional frame $\mathcal{D}$, the above equations induce a model of the simply-typed λ-calculus [3,20].

3.1 The Simple Model of ML Polymorphism

The simple model of ML polymorphism [23,22] defines the meaning of a ML-polymorphic expression in terms of type-indexed sets of denotations of its ground instances. It conservatively extends the type-frames semantics of the simply-typed λ-calculus [3,20] to accommodate polymorphism. It is a typed semantics, meaning that the denotations are given for derivable typing judgments of terms.

Ohori's model is quite intuitive. Consider the term $(\vdash \lambda x.x : \forall \alpha.\alpha \to \alpha)$. Any ground instance of this term (e.g., $\vdash \lambda x.x : \mathit{Int} \to \mathit{Int}$) has a meaning within a frame object (i.e., $D_{\mathit{Int} \to \mathit{Int}}$) of an appropriate frame $\mathcal{D}$. If the elements of $|D_{\tau \to \tau'}|$ are actually functions from $|D_\tau|$ to $|D_{\tau'}|$, then each of these ground instances is simply the identity function $id_{D_\tau} \in |D_{\tau \to \tau}|$. Accordingly, the meaning of $(\vdash \lambda x.x : \forall \alpha.\alpha \to \alpha)$ is just the set: $\{(\tau \to \tau, id_{D_\tau}) \mid \tau \in \mathsf{Type}\}$. We use product notation for the collection of these type-indexed sets: $\Pi\, \tau \in S.|D_\tau|$, each element of which is assumed to be a set function.

Defining an Ohori Model. This example illustrates the structure of Ohori's model: a semantics for the ground typings of an ML-polymorphic language may be extended conservatively to the full language; that is, a polymorphic term $(\Gamma \vdash e : \sigma)$ is defined by collecting the type-indexed denotations of its ground instances $\langle \tau, [\![\mathcal{A} \vdash e : \tau]\!] \rangle$. To accomplish this extension, one additional bit of machinery is necessary to calculate the ground instances of universal types, typing environments, and derivations. This is the subject of the remainder of this section.

We refer to a universal type σ containing no free type variables as a *closed type*. A typing environment Γ is *closed* when the type within each binding $(x : \sigma)$ is closed. A judgment $\Gamma \vdash M : \sigma$ is *closed* when Γ and σ are closed. A type derivation Δ is *closed* when the judgment at its root is closed. N.B., Δ being closed does not imply that its subderivations are closed. Given a substitution $\eta : \{\alpha_1, \ldots, \alpha_n\} \to \mathbf{T}_0$, its canonical extension η^* to a map from $\mathbf{T}_1$ to $\mathbf{T}_1$ is:

$$\begin{array}{ll} \eta^* \tau = \tau & \eta^* \alpha = \begin{cases} \eta\alpha & \text{if } \alpha \in \mathrm{dom}(\eta) \\ \alpha & \text{otherwise} \end{cases} \\ \eta^*(\gamma \to \gamma) = (\eta^* \gamma) \to (\eta^* \gamma) & \eta^*(\forall \beta . \sigma) = \forall \beta . (\eta_0^* \sigma) \text{ where } \eta_0 = \eta \setminus \beta \end{array}$$

Furthermore, we apply η^* to type environments and derivations as well:

$$\begin{array}{ll} \eta^*(x : \sigma, \Gamma) & = (x : \eta^* \sigma), \eta^* \Gamma \\ \eta^*\{\} & = \{\} \\ \eta^*\left(\frac{\Delta}{\Gamma \vdash M : \forall \bar{\alpha}.\gamma}\right) & = \frac{\eta_0^* \Delta}{(\eta_0^* \Gamma) \vdash M : \forall \bar{\alpha}.(\eta_0^* \gamma)} \text{ where } \eta_0 = \eta \setminus \bar{\alpha} \end{array}$$

First, we define the set of ground substitutions on a universal type:

$$\mathsf{Gr}(\sigma) \triangleq \{\eta \mid \eta : FV(\sigma) \to \mathsf{Type}\}$$

For a closed universal type $\sigma = (\forall \bar{\alpha}.\gamma)$ with (possibly empty) set of quantified variables $\bar{\alpha}$, the set of its *ground instances*, σ^{Type}, is:

$$\sigma^{\mathsf{Type}} \triangleq \{\eta^* \gamma \mid \ \eta \in \mathsf{Gr}(\gamma)\}$$

3.2 Solving Polymorphic-Recursive Equations

Frame models for the simply-typed lambda calculus consist of only data necessary to model application and abstraction. This data suffices for the simple model of ML polymorphism as the language modeled there does not include recursion [22]. Being recursive, the PR language requires more structure to model with frames and we extend the notion of type frame to accommodate recursion. In particular, the notion of a type frame is extended with structure including a partial order on the elements of frame sets, pointedness of frame objects, and continuous functions that preserve order and limits. This same methodology has been employed to specify Haskell's mixture of lazy and eager evaluation [7].

We return to the question posed in Section 1 at (†): how—or rather, *where*—do we solve equations like:

$$\mathit{size} \ = \ \lambda x.\, \textbf{if}\ \mathit{isNil}\ x\ \textbf{then}\ 0\ \textbf{else}\ 1 + 2 * \mathit{size}\ (\mathit{stl}\ x)$$

As it turns out, if frame $\mathcal{D}$ is such that its frame objects, D_τ, are pointed cpos, then the denotations of polymorphic types in Ohori's model (i.e., the indexed sets $(\Pi\, \tau \in S. |D_\tau|)$) may be extended to pointed cpos[2] as well. This idea is made formal in Theorem 1 below. Within these new cpo structures, we can find appropriate solutions to polymorphic-recursive equations and define the semantics of PR (as we do below in Section 3.3).

[2] Technically, this is an ω-cpo; that is, every ascending chain possesses a lub. For a cpo, every directed set has a lub. N.B., every cpo is an ω-cpo, but not vice-versa.

Frames for Polymorphic Recursion. We introduce the following terminology for such frames with a pointed cpo structure: A *pcpo frame* is a frame $\langle \mathcal{D}, \bullet, \sqsubseteq, \sqcup, \bot \rangle$ in which each $D_\tau \in \mathcal{D}$ is a pointed, complete, partial order w.r.t. $\bot_\tau$, $\sqsubseteq_\tau$, and $\sqcup_\tau$. For a given pcpo frame $\mathcal{D}$, a type-indexed set $(\Pi\, \tau \in S.|D_\tau|)$ is uniquely determined by the set of types S, and accordingly we introduce the abbreviation $P_S \triangleq (\Pi\, \tau \in S.|D_\tau|)$. If S is the singleton set $\{\tau\}$, we write P_τ. A family of application operators $\bullet$ is defined for the P_τ in the obvious manner in terms of the $\bullet$ operators in $\mathcal{D}$. Theorem 1 demonstrates that the collection of P_S are the frame objects in a pcpo frame; we will refer to this frame hereafter as $\mathcal{P}$. The language PR is defined in the frame $\mathcal{P}$ below in Section 3.3.

Theorem 1. *Let $\langle \mathcal{D}, \bullet, \sqsubseteq, \sqcup, \bot \rangle$ be a pcpo-frame and $S \subseteq \mathsf{Type}$ be a set of ground type expressions. Then, P_S is a pointed cpo where:*

- *for any $f, g \in P_S$, $f \sqsubseteq_S g \Leftrightarrow$ for all $\tau \in S$, $f\tau \sqsubseteq_\tau g\tau$, and*
- *the bottom element is $\bot_S \triangleq \{\langle \tau, \bot_\tau \rangle \mid \tau \in S\}$*
- *Let $\sqcup_\tau$ be the lub operator within the $\mathcal{D}$ frame object D_τ. Then the least upper bound of an ascending chain $\{X_i\} \subseteq |P_S|$ is:*
$$\bigsqcup_S X_i \triangleq \{\langle \tau, u_\tau \rangle \mid \tau \in S, u_\tau = \bigsqcup_\tau (X_i \tau)\}$$

Proof. To show: $\sqsubseteq_S$ and $\bot_S$ define a pointed, complete partial order on P_S. Assume $X, X_i \in P_S$. That $\sqsubseteq_S$ is reflexive, anti-symmetric, and transitive follows directly from the fact the each $\sqsubseteq_\tau$ is so. Similarly, $\bot_S$ is the least element of P_S because, for any $\tau \in \mathsf{Type}$, so is $\bot_S\ \tau = \bot_\tau$. Let $X_0 \sqsubseteq_S X_1 \sqsubseteq_S \cdots$ be a directed chain in P_S, then it remains to show that $\{\langle \tau, u_\tau \rangle \mid \tau \in S, u_\tau = \bigsqcup_\tau (X_i \tau)\}$ is the lub of $\{X_i\}$. Define $U = \{\langle \tau, u_\tau \rangle \mid \tau \in S, u_\tau = \bigsqcup_\tau (X_i \tau)\}$. It is clear from the definition of U and $\sqsubseteq_S$ that U is in P_S and is an upper limit of $\{X_i\}$. Suppose that $V \in P_S$ is an upper bound of $\{X_i\}$. Then, $X_i \tau \sqsubseteq_\tau V\tau$ for every type $\tau \in \mathsf{Type}$. $\therefore \bigsqcup_\tau (X_i \tau) \sqsubseteq_\tau V\tau$ for every type $\tau \in \mathsf{Type}$. By the definition of U, $U\tau \sqsubseteq_\tau V\tau$ for every type $\tau \in \mathsf{Type}$ and, hence, $U \sqsubseteq_S V$. $\therefore U$ is the least such upper limit, justifying the definition of $\bigsqcup_S X_i \triangleq U$. □

Because P_S is a pointed cpo, we may define continuous functions and least fixed points over it in the standard way [3]. That is, a function $f : P_S \to P_T$ is *continuous* when $f(\bigsqcup_S X_i) = \bigsqcup_T (f\, X_i)$ for all directed chains $\{X_i\}$ in P_S. Then, $\mathit{fix}(f) = \bigsqcup_S (f^n\ \bot_S)$ for $n < \omega$ given a continuous endofunction $f : P_S \to P_S$. We assume that every continuous function between frame objects in frame $\mathcal{P}$ is representable in $\mathcal{P}$ and that $\mathcal{P}$ is extensional.

3.3 The Frame Semantics of PR

What remains to be seen is the definition of the PR language in terms of the pcpo frame $\mathcal{P}$. First, we consider the denotation of universal types and then the semantics of PR is given.

Denotations of Types. A closed universal type is denoted by the type-indexed set of its ground instances: $[\![\sigma]\!] = \Pi\, \tau \in (\sigma^{\mathsf{Type}}).|D_\tau|$ for closed σ.

Denotations of Terms. The denotation of PR is defined on term derivations. Recall that PR is syntax-oriented; the coherence of the semantic equations follows because the derivations are unique (modulo the order of application of the **GEN** rule). Let Δ be a derivation of $(\Gamma \vdash e : \sigma)$ and $\eta \in \mathsf{Gr}(\sigma)$, then the signature of the semantics is $[\![\Delta]\!]\eta : \mathsf{Env}(\eta^*\Gamma) \to [\![\eta^*\sigma]\!]$. N.B., $\eta^*\Gamma$ is a closed type assignment, and $\mathsf{Env}(\eta^*\Gamma)$ are the environments compatible with that closed type assignment. The semantic equations are defined by induction on derivation trees; this is the subject of the remainder of this section.

GEN. Let $\sigma = \forall\alpha_1.\dots\forall\alpha_n.\gamma$, $\eta \in \mathsf{Gr}(\sigma)$, and $\rho \in \mathsf{Env}(\eta^*\Gamma)$. Let $\oplus$ be right-biased environment extension:

$$(\eta \oplus \eta')\alpha = \begin{cases} \eta'\alpha & \text{if } \alpha \in \mathsf{dom}(\eta') \\ \eta\alpha & \text{otherwise} \end{cases}$$

Note that $S \triangleq \sigma^{\mathsf{Type}}$ may be written in terms of extensions to η as:

$$S = \{\tau \in \mathsf{Type} \mid \eta_\tau : \{\alpha_1, \dots, \alpha_n\} \to \mathsf{Type},\ \tau = (\eta \oplus \eta_\tau)^*\gamma\}$$

and, furthermore for any $\tau \in S$, the extension η_τ such that $\tau = (\eta \oplus \eta_\tau)^*\gamma$ is unique. We may therefore define the semantics of a GEN rule application as:

$$[\![\Gamma \vdash e : \sigma]\!]\eta\rho = \bigcup_{\tau \in S} \left([\![\Gamma \vdash e : \gamma]\!](\eta \oplus \eta_\tau)\rho \right) \tag{1}$$

VAR. An auxiliary function, $\iota_\tau\, x = \{\langle \tau, x \rangle\}$, injects D_τ into P_τ. N.B., that if $x \in |D_\tau|$, then $\iota_\tau\, x \subseteq (\Pi\, \tau \in S.|D_\tau|)$ for any $S \subseteq \mathsf{Type}$ such that $\tau \in S$. The binding of a variable x in ρ is a type-indexed set and the denotation of x is a component of this set:

$$[\![\Gamma \vdash x : \gamma]\!]\eta\rho = \iota_\tau(\rho\, x\, \tau), \text{ where } \tau = \eta^*\gamma \tag{2}$$

The definition in (2) determines the correct component of $[\![\forall\alpha.\mathit{Seq}(\alpha) \to \mathit{Int}]\!]$, τ, from the ground substitution η and the open type γ. The *size* example illustrates why this definition works. Consider the VAR application in the example derivation at the end of Section 2; recall the judgment at the root is: $\Gamma_2 \vdash \mathit{size} : \mathit{Seq}(\alpha \times \alpha) \to \mathit{Int}$. Let $\eta \in \mathsf{Gr}(\mathit{Seq}(\alpha \times \alpha) \to \mathit{Int})$ and $\rho \in \mathsf{Env}(\eta^*\Gamma_2)$, and assume $\eta\alpha = \mathit{Bool}$. Note that $(\rho\, \mathit{size}) \in (\Pi\, \tau \in \sigma^{\mathsf{Type}}.|D_\tau|)$ for $\sigma = \forall\alpha.\mathit{Seq}(\alpha) \to \mathit{Int}$. The component of $(\rho\, \mathit{size})$ denoting x is found at the ground type $\eta^*(\mathit{Seq}(\alpha \times \alpha) \to \mathit{Int}) = \mathit{Seq}(\mathit{Bool} \times \mathit{Bool}) \to \mathit{Int}$.

PFIX. Assuming $\eta \in \mathsf{Gr}(\gamma)$ and $\rho \in \mathsf{Env}(\eta^*\Gamma)$, let S and $f_S : P_S \to P_S$ be the following set and function, respectively:

$$\begin{aligned} S &= \{\tau \in \mathsf{Type} \mid \tau = \eta^*\gamma'\} \\ f_S &= d \mapsto [\![\Gamma, x : \forall\bar{\alpha}.\gamma' \vdash M : \gamma']\!]\eta(\rho[x \mapsto d]) \end{aligned}$$

By Lemma 1 (below), f_S is continuous, and we make the additional assumption that f_S is representable in $\mathcal{P}$. Then, polymorphic fixpoints are defined as:

$$[\![\Gamma \vdash \mathrm{pfix}\, x.M : \gamma]\!]\eta\rho = \bigsqcup\nolimits_S (f^n \perp_S) \text{ for } n < \omega \tag{3}$$

ABS. Let $\tau = \eta^*\gamma$ and $\tau' = \eta^*\gamma'$ and f be the function from $\mathcal{P}_\tau$ to $\mathcal{P}_{\tau'}$:

$$f = d \mapsto [\![\Gamma, x : \gamma \vdash M : \gamma']\!]\eta(\rho[x \mapsto d])$$

By Lemma 1 (below), f is continuous, and we assume that f is representable in $\mathcal{P}$. Then, lambda abstraction is defined as:

$$[\![\Gamma \vdash \lambda x.M : \gamma \to \gamma']\!]\eta\rho = f \tag{4}$$

APP. Let $\tau = \eta^*\gamma$, $\tau' = \eta^*\gamma'$. Then the semantics of application is simply:

$$[\![\Gamma \vdash MN : \gamma']\!]\eta\rho = ([\![\Gamma \vdash M : \gamma \to \gamma']\!]\eta\rho) \bullet_{\tau,\tau'} ([\![\Gamma \vdash N : \gamma]\!]\eta\rho) \tag{5}$$

N.B., the application operator, $\bullet_{\tau,\tau'}$, is from the objects $\mathcal{P}_{(\tau \to \tau')}$ and $\mathcal{P}_\tau$.

LET. The semantics of *let* is defined conventionally:

$$[\![\Gamma \vdash (\text{let } x = N \text{ in } M) : \sigma]\!]\eta\rho = [\![\Gamma, x : \sigma' \vdash M : \sigma]\!]\eta(\rho[x \mapsto [\![\Gamma \vdash N : \sigma']\!]\eta\rho]) \tag{6}$$

The semantic equations for PFIX and ABS rely on Lemma 1 which states that functions defined in terms of the semantic function are continuous. Lemma 1, or something very much like it, is part of any conventional cpo semantics of λ-calculi. Its proof follows along the lines of what one would find in a semantics textbook (e.g., see Gunter 1992, lemma 4.19, page 130).

Lemma 1. *For any closed term* $(\Gamma, x : \forall\bar{\alpha}.\gamma' \vdash M : \gamma)$*, the following function from* P_S *to* P_T *is continuous:* $f = (d \in |P_S|) \mapsto [\![\Gamma, x : \forall\bar{\alpha}.\gamma' \vdash M : \gamma]\!]\eta\rho[x \mapsto d])$.

Proof. By induction on M. Let $\Gamma' = \Gamma, x : \forall\bar{\alpha}.\gamma$. Each case is straightforward so we show only the case for $M = \text{pfix}y.\ M'$.

To show: f is monotonic. Let $e, e' \in |P_S|$ such that $e \sqsubseteq_S e'$. Define $d_i \in |P_S|$ for $i < \omega$ as $d_0 = \bot_S$ and $d_{i+1} = [\![\Gamma', y : \sigma \vdash M' : \gamma]\!]\eta\rho[y \mapsto d_i]$. Note that, by definition, $P_T = [\![\eta^*\gamma]\!]$. Then,

$$\begin{aligned}
&[\![\Gamma' \vdash \text{pfix } y.M' : \gamma]\!]\eta\rho[x \mapsto e] \\
&\quad = \textstyle\bigsqcup_T([\![\Gamma', y : \sigma \vdash M' : \gamma]\!]\eta\rho[x \mapsto e][y \mapsto d_i]) && \{\text{defn.}\} \\
&\quad \sqsubseteq_T \textstyle\bigsqcup_T([\![\Gamma', y : \sigma \vdash M' : \gamma]\!]\eta\rho[x \mapsto e'][y \mapsto d_i]) && \{\text{ind. hyp.}\} \\
&\quad = [\![\Gamma' \vdash \text{pfix } y.M' : \gamma]\!]\eta\rho[x \mapsto e'] && \{\text{defn.}\}
\end{aligned}$$

To show: f is preserves limits. Let $P_X = [\![\eta^*\forall\bar{\alpha}.\gamma']\!]$ and $x_i \in |P_X|$ be such that $x_i \sqsubseteq_X x_{i+1}$. Then,

$$\begin{aligned}
&[\![\Gamma' \vdash \text{pfix } y.M' : \gamma]\!]\eta\rho[x \mapsto \sqcup_X x_i] \\
&\quad = \textstyle\bigsqcup_T([\![\Gamma', y : \sigma \vdash M' : \gamma]\!]\eta\rho[x \mapsto \sqcup_X x_i][y \mapsto d_i]) && \{\text{defn.}\} \\
&\quad = \textstyle\bigsqcup_T \bigsqcup_X([\![\Gamma', y : \sigma \vdash M' : \gamma]\!]\eta\rho[x \mapsto x_i][y \mapsto d_i]) && \{\text{ind. hyp.}\} \\
&\quad = \textstyle\bigsqcup_X \bigsqcup_T([\![\Gamma', y : \sigma \vdash M' : \gamma]\!]\eta\rho[x \mapsto x_i][y \mapsto d_i]) && \{\text{exchange}\} \\
&\quad = \textstyle\bigsqcup_X([\![\Gamma' \vdash \text{pfix } y.M' : \gamma]\!]\eta\rho[x \mapsto x_i]) && \{\text{defn.}\}
\end{aligned}$$

Here, {exchange} refers to what is frequently known as the *exchange lemma.* This states that, given a monotone function, $f : P \times Q \rightarrow D$, for directed sets P and Q and cpo D, that the ordering of the limits does not matter:

$$\bigsqcup_{x \in P} \bigsqcup_{y \in Q} f(x,y) = \bigsqcup_{y \in Q} \bigsqcup_{x \in P} f(x,y)$$

An exchange lemma holds for $\mathcal{P}$; its proof is exactly the same as that of (Gunter 1992, Lemma 4.9, page 118). □

4 Application to Haskell Type Classes

This work is part of an effort[3] to develop a formal basis for reasoning about the Haskell core language known as Haskell 98 [25]. The published work in this endeavor focuses on mixed evaluation (i.e., combined lazy and eager evaluation) in Haskell [7,6] and its semantic and logical consequences. The original interest in the simple model of ML polymorphism stems from the observation that its "type awareness" provides precisely what one needs to specify type classes in Haskell; the rest of this section presents a high-level overview of this insight. The initial presentation of polymorphic recursion [21] developed a denotational semantics based on the ideal model of recursive types [17]. Cousot [1] formulated a hierarchy of type systems—including Mycroft's—in terms of a lattice of abstract interpretations of the untyped lambda calculus. However, neither of these potential starting points—i.e., the ideal model or abstract interpretation—seem to fit quite as well to type classes as does the Ohori framework.

Type classes in Haskell are an overloading mechanism allowing variables to be interpreted differently at different types. The *Eq* class (shown in part below) defines equality (==); it is the primitive *eqInt* on instance (*Int* → *Int* → *Bool*). Equality on pairs of type (a, b) is inherited from equality on a and b instances; in the last instance declaration below, "$x{=}{=}u$" and "$y{=}{=}v$" are equality on types a and b, respectively:

```
class Eq a where
  (==) :: a → a → Bool
instance Eq Int where
  (==) = eqInt
instance (Eq a, Eq b) ⇒ Eq (a, b) where
  (x, y)==(u, v)  = (x==u)&&(y==v)
```

[3] The *Programatica* project explores the use of Haskell 98 in formal methods. See `www.cse.ogi.edu/PacSoft/projects/programatica` for further details.

Polymorphic recursion first entered Haskell through its type class system when expert practitioners noticed that polymorphic recursive functions could be expressed via "kludges" such as the implementation of *size* below [12]:

```
size = size′ ()
class Size d where
   size′ :: d → Seq a → Int
instance Size () where
   size′ p Nil           = 0
   size′ p (SCons x xs)  = 1 + 2 ∗ size′ p xs
```

The method "*size′*" contains a dummy parameter but is otherwise identical to *size*. Because type inference is undecidable in the presence of polymorphic recursion [8], the type of *size′* must be declared explicitly and the kludge uses the type declaration of the *size′* method for the same purpose as the original explicit type declaration of *size*. The type of the dummy variable—d in the type signature for *size′*—is the overloaded type here. The Haskell type system considers the type variable a in the *size′* method signature as quantified (analogously to the universally typed recursive binding of x in the PFIX rule). Creating a dummy instance of *Size* at unit type () allows the definition of the polymorphic recursive function *size* by simply applying *size′* to the unit value ().

Without polymorphic recursion, overloading in Haskell 98 may be handled via partial evaluation [14]; that is, the finitely many class instances necessary within a Haskell program may be determined statically and inlined within the program itself. With polymorphic recursion, the number of such instances is potentially unbounded; consider the function $\mathit{foo}\ x = x{=}{=}x\ \&\&\ \mathit{foo}\ (x, x)$ which will require an unbounded number of implementations of $==$.

Within a suitable frame $\mathcal{P}$ from Section 3, however, a type class may be viewed as the set of its ground instances:

$$[\![\mathsf{Eq}]\!] = \{\langle \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Bool}, \mathit{eqInt}\rangle, \langle \mathit{Float} \rightarrow \mathit{Float} \rightarrow \mathit{Bool}, \mathit{eqFloat}\rangle, \ldots\}$$

This denotation should appeal to the intuitions of Haskell programmers, because, according to this view, a type class is merely the type-indexed set of its instances—what would be called in Haskell terminology a *dictionary* [4]. It is important to note that, while potentially infinite dictionaries foil the use of partial evaluation as an implementation technique for type classes in general, they pose no problem for the denotational model for type classes outlined in this section.

Models and implementation techniques for type classes have been considered for many years now; a representative, albeit quite non-exhaustive, sampling of this work is [29,28,15,4,27]. All existing models of type classes have one thing in common—they are based on the translation of the source language into an intermediate form for which a model or implementation is known. Thatte [28] translates a first-order λ-calculus extended with a notation for classes and instances (called "OML") into a second-order lambda calculus with a special **typecase** construct for examining type structure (called "OXML"). OXML is then given an interpreter and OML is defined indirectly via its translation into OXML. The *Eq* class example above would be translated as:

$$\begin{aligned} (==) \; &= \Lambda t.\, \textbf{typecase}\; t\; \textbf{of}\; \{\; \mathit{Int}\colon \mathit{eqInt}\;;\; a{\times}b\colon \mathit{eqPair}\langle a\rangle\langle b\rangle\; \} \\ \mathit{eqPair} \; &= \Lambda a.\, \Lambda b.\, \lambda(x{:}a, y{:}b).\, \lambda(u{:}a, v{:}b).\, (==\; \langle a\rangle\; x\; u)\;\&\&\;(==\; \langle b\rangle\; y\; v) \end{aligned}$$

Here, Λ and $\langle - \rangle$ represent type abstraction and application in OXML, respectively. The translation of (==) takes the type t at which it is applied and dynamically determines the appropriate method implementation. If $t = a \times b$, then this involves creating instances of (==) at types a and b (as is done in the body of *eqPair*). Thatte's semantics is similar to the implementation of Haskell classes via the *dictionary-passing translation* (DPT) in which functions with overloaded types are translated to functions taking an additional dictionary parameter [29,15,4,27].

In contrast to existing models, the approach outlined in this section is direct: no intermediate translation of the source language is required. Furthermore, the "dictionary" denotations of classes in $\mathcal{P}$ require no dynamic examination of type structure. The denotation of *Eq* above, for example, contains all instances of *Eq*; no **typecase** construct or on-the-fly dictionary construction is required.

5 Conclusions and Future Directions

This article demonstrates an approach to modeling polymorphic recursion in functional languages. These models require a fundamental change to the substance of denotations rather than to their form and this shift is recorded in the construction of the $\mathcal{P}$ type frame (Theorem 1). Types may still be modeled by cpos and recursion by fixed point calculations as in traditional denotational semantics, but the underlying structure of those cpos changes to include type information. This "type awareness" accommodates the additional expressiveness required to model polymorphic recursion.

The Girard-Reynolds calculus uses abstraction over an (impredicative) universe of types. Hindley-Milner polymorphism is considerably more restrictive in that it only allows abstraction and type quantification over values and base types, respectively. Ohori's insight was that the relative restrictiveness of Hindley-Milner admits a predicative semantics; in fact, any frame model of the simply-typed lambda calculus may be conservatively extended to a model of Hindley-Milner. This article demonstrates that this extension may continue one step beyond Hindley-Milner to a predicative model of first-order polymorphic recursion and, furthermore, that this extension is straightforward (albeit non-trivial).

This work reports progress in an effort to establish a formal semantics for the entire Haskell 98 language starting from Ohori's simple model of ML polymorphism [23,22]. Extensions to the Ohori model have already been explored for characterizing Haskell's surprisingly complex mixed evaluation [7]. Obviously, any such Haskell semantics must account for polymorphic recursion for the simple reason that Haskell allows polymorphic recursive definitions. As was described in Section 4, the type awareness of the $\mathcal{P}$ type frame suggests a natural denotational model of type classes, the precise details of which are left to a sequel.

Acknowledgements

The author wishes to thank Dick Kieburtz, Brian Dunphy, and Mark Jones for their encouragement and for numerous discussions on aspects of polymorphic recursion, its relationship to the semantics of Haskell, and feedback on this article.

References

1. P. Cousot. Types as abstract interpretations, invited paper. In *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–331, Paris, France, January 1997. ACM Press, New York, NY.
2. Jean-Yves Girard. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
3. Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.
4. Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.
5. Robert Harper and John C. Mitchell. On the type structure of standard ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):211–252, 1993.
6. William Harrison, Timothy Sheard, and James Hook. Fine control of demand in Haskell. In *6th International Conference on the Mathematics of Program Construction, Dagstuhl, Germany*, volume 2386 of *Lecture Notes in Computer Science*, pages 68–93. Springer-Verlag, 2002.
7. William L. Harrison and Richard B. Kieburtz. The logic of demand in Haskell. *Journal of Functional Programming*, 15(5), 2005.
8. Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.
9. Roger J. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.
10. Ralf Hinze. A new approach to generic functional programming. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132, 2000.
11. Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming: Advanced Lectures*, number 2793 in Lecture Notes in Computer Science, pages 1–56. Springer-Verlag, 2003.
12. Mark Jones. Private communication.
13. Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 52–61. ACM Press, 1993.

14. Mark P. Jones. Dictionary-free overloading by partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, Florida, June 1994.
15. Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, England, 1994.
16. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.
17. D. B. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2), 1984.
18. Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Science*, 17(3):348–375, 1978.
19. J. C. Mitchell and R. Harper. The essence of ML. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages (POPL 1988)*, pages 28–46, 1988.
20. John C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, MA, Third edition, 2000.
21. Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228, Toulouse, France, April 1984. Springer.
22. Atsushi Ohori. A Simple Semantics for ML Polymorphism. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture, Imperial College, London*, pages 281–292, September 1989.
23. Atsushi Ohori. *A Study of Semantics, Types, and Languages for Databases and Object-oriented Programming*. PhD thesis, University of Pennsylvania, 1989.
24. Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
25. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, the Revised Report*. Cambridge University Press, April 2003.
26. John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, April 1974.
27. Peter J. Stuckey and Martin Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2004. To appear in ACM Transactions on Programming Languages and Systems.
28. Satish R. Thatte. Semantics of type classes revisited. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 208–219. ACM Press, 1994.
29. Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, 1989.

Symbolic Execution with Separation Logic

Josh Berdine[1], Cristiano Calcagno[2], and Peter W. O'Hearn[1]

[1] Queen Mary, University of London
[2] Imperial College, London

Abstract. We describe a sound method for automatically proving Hoare triples for loop-free code in Separation Logic, for certain preconditions and postconditions (symbolic heaps). The method uses a form of symbolic execution, a decidable proof theory for symbolic heaps, and extraction of frame axioms from incomplete proofs. This is a precursor to the use of the logic in automatic specification checking, program analysis, and model checking.

1 Introduction

Separation Logic has provided an approach to reasoning about programs with pointers that often leads to simpler specifications and program proofs than previous formalisms [12]. This paper is part of a project attempting to transfer the simplicity of the by-hand proofs to a fully automatic setting.

We describe a method for proving Hoare triples for loop-free code, by a form of symbolic execution, for certain (restricted) preconditions and postconditions. It is not our intention here to try to show that the method is useful, just to say what it is, and establish its soundness. This is a necessary precursor to further possible developments on using Separation Logic in:

- *Automatic Specification Checking*, where one takes an annotated program (with preconditions, postconditions and loop invariants) and chops it into triples for loop-free code in the usual way;
- *Program Analysis*, where one uses fixed-point calculations to remove or reduce the need for annotations; and
- *Software Model Checking.*

The algorithms described here are, in fact, part an experimental tool of the first variety, Smallfoot. Smallfoot itself is described separately in a companion paper [2]; here we confine ourselves to the technical problems lying at its core. Of course, program analysis and model checking raise further problems – especially, the structure of our "abstract" domain and the right choice of widening operators [3] – and further work is under way on these.

There are three main issues that we consider.

1. How to construe application of Separation Logic proof rules as symbolic execution. The basic idea can be seen in the axiom

$$\{A * x{\mapsto}[f\colon y]\}\ x{\to}f{:=}z\ \{A * x{\mapsto}[f\colon z]\}$$

where the precondition is updated in-place, in a way that mirrors the imperative update of the actual heap that occurs during program execution. The separating conjunction, $*$, short-circuits the need for a global alias check in this axiom. $A * x \mapsto [f\colon y]$ says that the heap can be partitioned into a single cell x, that points to (has contents) a record with y in its f field, and the rest of the heap, where A holds. We know that A will continue to hold in the rest of the heap if we update x, because x is not in A's part of the heap.

There are two restrictions on assertions which make the symbolic execution view tenable. First, we restrict to a format of the form $B \wedge S$ where B is a pure boolean formula and S is a $*$-combination of heap predicates. We think of these assertions as "symbolic heaps"; the format makes the analogy with the in-place aspect of concrete heaps apparent. Second, the preconditions and postconditions do not describe the detailed contents of data structures, but rather describe shapes (in roughly the sense of the term used in shape analysis). Beyond the basic primitives of Separation Logic, Smallfoot at this point includes several hardwired shape predicates for: singly- and doubly-linked lists, xor-linked lists, and trees. Here we describe our results for singly-linked lists and trees only.

2. How to discharge entailments $A \vdash B$ between symbolic heaps. We give a decidable proof theory for the assertions in our language.

One key issue is how to account for entailments that would normally require induction. To see the issue, consider a program for appending two lists. When you get to the end of the first list you link it up to the second. At this point to prove the program requires showing an entailment

$$\mathsf{ls}(x,t) * t \mapsto [n\colon y] * \mathsf{ls}(y,\mathsf{nil}) \vdash \mathsf{ls}(x,\mathsf{nil})$$

where we have a list segment from x to t, a single node t, and a further segment (the second list) from y up to nil. The entailment itself does not follow at once from simple unwinding of an inductive definition of list segments. In the metatheory it is proven by induction, and in our proof theory it will be handled using rules that are consequences of induction but that are themselves non-inductive in character.

In [1] we showed decidability of a fragment of the assertion language of this paper, concentrating on list segments. Here we give a new proof procedure, which appears to be less fragile in the face of extension than the model-theoretic procedure of [1], since if the fragment is extended with additional formulæ, then the decision procedure of [1] remains complete but potentially becomes unsound, while the present proof theory remains sound but potentially becomes incomplete. Additionally, and crucially, it supports inference of frame axioms.

3. Inference of Frame Axioms. Separation Logic allows specifications to be kept small because it avoids the need to state frame axioms, which describe the portions of the heap not altered by a command [10]. To see the issue, consider a specification

$$\{\mathsf{tree}(p)\}\ \mathrm{disp\_tree}(p)\ \{\mathsf{emp}\}$$

for disposing a tree, which just says that if you have a tree (and nothing else) and you dispose it, then there is nothing left. When verifying a recursive procedure for disposing a tree there will be recursive calls for disposing subtrees. The problem is that, generally, a precondition at a call site will not match that for the procedure due to extra heap around. For example, at the site of a call disp_tree(i) to dispose the left subtree we might have a root pointer p and the right subtree j as well as the left subtree – $p \mapsto [l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)\}$ – while the precondition for the overall procedure specification expects only a single tree.

Separation Logic has a proof rule, the Frame Rule, which allows us to resolve this mismatch. It allows us the first step in the inference:

$$\dfrac{\dfrac{\{\mathsf{tree}(i)\}\ \mathrm{disp\_tree}(i)\ \{\mathsf{emp}\}}{\{p \mapsto [l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)\}\ \mathrm{disp\_tree}(i)\ \{p \mapsto [l\colon i, r\colon j] * \mathsf{emp} * \mathsf{tree}(j)\}}}{\{p \mapsto [l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)\}\ \mathrm{disp\_tree}(i)\ \{p \mapsto [l\colon i, r\colon j] * \mathsf{tree}(j)\}}$$

To automatically generate proof steps like this we need some way to infer frame axioms, the leftover parts (in this case $p \mapsto [l\colon i, r\colon j] * \mathsf{tree}(j)$). Sometimes, this leftover part can be found by simple pattern matching, but often not. In this paper we describe a novel method of extracting frame axioms from incomplete proofs in our proof theory for entailments. A failed proof can identify the "left-over" part which, were you to add it in, would complete the proof, and we show how this can furnish us with a sound choice of frame axiom.

The notion of symbolic execution presented in this paper is, in a general sense, similar in spirit to what one obtains in Shape Analysis or PALE [14, 7]. However, there are nontrivial differences in the specifics. In particular, we have been unsuccessful in attempts to compositionally translate Separation Logic into either PALE's assertion language or into a shape analysis; the difficulty lies in treating the separating conjunction connective. And this is the key to employing the frame rule, which is responsible for Separation Logic's small specifications of procedures. So it seems sensible to attempt to describe symbolic execution for Separation Logic directly, in its own terms.

2 Symbolic Heaps

The concrete heap models assume a fixed finite collection Fields, and disjoint sets Loc of locations, Val of non-addressable values, with $\mathsf{nil} \in$ Val. We then set:

$$\begin{aligned} \text{Heaps} &\overset{\text{def}}{=} \text{Loc} \overset{\text{fin}}{\rightharpoonup} (\text{Fields} \to \text{Val} \cup \text{Loc}) \\ \text{Stacks} &\overset{\text{def}}{=} \text{Var} \to \text{Val} \cup \text{Loc} \end{aligned}$$

As a language for reasoning about these models we consider certain pure (heap independent) and spatial (heap dependent) assertions.

$x, y, \ldots \in \text{Var}$	variables
$E ::= \mathsf{nil} \mid x$	expressions
$P ::= E{=}E \mid \neg P$	simple pure formulæ
$\Pi ::= \mathsf{true} \mid P \mid \Pi \wedge \Pi$	pure formulæ
$f, f_i, \ldots \in \text{Fields}$	fields
$\rho ::= f_1\colon E_1, \ldots, f_k\colon E_k$	record expressions
$S ::= E \mapsto [\rho]$	simple spatial formulæ

$$\begin{array}{ll} \Sigma ::= \mathsf{emp} \mid S \mid \Sigma * \Sigma & \text{spatial formulæ} \\ \Pi \mathrel{¦} \Sigma & \text{symbolic heaps} \end{array}$$

Symbolic heaps are pairs $\Pi \mathrel{¦} \Sigma$ where Π is essentially an $\wedge$-separated sequence of pure formulæ, and Σ a $*$-separated sequence of simple spatial formulæ.[1] The pure part here is oriented to stating facts about pointer programs, where we will use equality with nil to indicate a situation where we do not have a pointer. Other subsets of boolean logic could be considered in other situations.

In this heap model a location maps to a record of values. The formula $E\mapsto[\rho]$ can mention any number of fields in ρ, and the values of the remaining fields are implicitly existentially quantified. This allows us to write specifications which do not mention fields whose values we do not care about.

The semantics is given by a forcing relation $s, h \vDash A$ where $s \in$ Stacks, $h \in$ Heaps, and A is a pure assertion, spatial assertion, or symbolic heap. $h = h_0 * h_1$ indicates that the domains of h_0 and h_1 are disjoint, and h is their graph union.

$$[\![x]\!]s \overset{\text{def}}{=} s(x) \qquad\qquad [\![\mathsf{nil}]\!]s \overset{\text{def}}{=} \text{nil}$$

$$\begin{array}{lll}
s, h \vDash E_1{=}E_2 & \overset{\text{def}}{\text{iff}} & [\![E_1]\!]s = [\![E_2]\!]s \\
s, h \vDash \neg P & \overset{\text{def}}{\text{iff}} & s, h \nvDash P \\
s, h \vDash \mathsf{true} & & \text{always} \\
s, h \vDash \Pi_0 \wedge \Pi_1 & \overset{\text{def}}{\text{iff}} & s, h \vDash \Pi_0 \text{ and } s, h \vDash \Pi_1 \\
s, h \vDash E_0\mapsto[f_1{:}\, E_1, \ldots, f_k{:}\, E_k] & \overset{\text{def}}{\text{iff}} & h = [[\![E_0]\!]s \to r] \text{ where } r(f_i) = [\![E_i]\!]s \text{ for } i \in 1..k \\
s, h \vDash \mathsf{emp} & \overset{\text{def}}{\text{iff}} & h = \varnothing \\
s, h \vDash \Sigma_0 * \Sigma_1 & \overset{\text{def}}{\text{iff}} & \exists h_0 h_1.\, h = h_0 * h_1 \text{ and } s, h_0 \vDash \Sigma_0 \text{ and } s, h_1 \vDash \Sigma_1 \\
s, h \vDash \Pi \mathrel{¦} \Sigma & \overset{\text{def}}{\text{iff}} & s, h \vDash \Pi \text{ and } s, h \vDash \Sigma
\end{array}$$

To reason about pointer programs one typically needs predicates that describe inductive properties of the heap. We describe two of the predicates (adding to the simple spatial formulæ) that we have experimented with in Smallfoot.

2.1 Trees

We describe a model of binary trees where each internal node has fields l, r for the left and right subtrees. The empty tree is given by nil. What we require is that $\mathsf{tree}(E)$ is the least (logically strongest) predicate satisfying:

$$\mathsf{tree}(E) \iff (E = \mathsf{nil} \wedge \mathsf{emp}) \vee (\exists x, y.\, E\mapsto[l{:}\, x, r{:}\, y] \;*\; \mathsf{tree}(x) \;*\; \mathsf{tree}(y))$$

where x and y are fresh. The use of the $*$ between $E\mapsto[l{:}\, x, r{:}\, y]$ and the two subtrees ensures that there are no cycles, and the $*$ between the subtrees ensures that there is no sharing; it is not a DAG.

[1] Note that we abbreviate $\neg(E_1{=}E_2)$ as $E_1{\neq}E_2$ and $\mathsf{true} \mathrel{¦} \Sigma$ as Σ, and use $\equiv$ to denote "syntactic" equality of formulæ, which are considered up to symmetry of $=$, permutations across $\wedge$ and $*$, *e.g.*, $\Pi \wedge P \wedge P' \equiv \Pi \wedge P' \wedge P$, involutivity of negation, and unit laws for true and emp. We use notation treating formulæ as sets of simple formulæ, *e.g.*, writing $P \in \Pi$ for $\Pi \equiv P \wedge \Pi'$ for some Π'.

The way that the record notation works allows this definition to apply to any heap model that contains at least l and r fields. In case there are further fields, say a field d for the data component of a node, the definition is independent of what the specific values are in those fields.

Our description of this predicate is not entirely formal, because we do not have existential quantification, disjunction, or recursive definitions in our fragment. However, what we are doing is defining a new simple spatial formula (extending syntactic category S above), and we are free to do that in the metatheory. A longer-winded way to view this, as a semantic definition, is to say that it is the least predicate such that

$s, h \vDash \mathsf{tree}(E)$ holds if and only if

1. $s, h \vDash E = \mathsf{nil} \wedge \mathsf{emp}$, or
2. ℓ_x, ℓ_y exist where $(s \mid x{\to}\ell_x, y{\to}\ell_y), h \vDash E{\mapsto}[l\colon x, r\colon y] \ast \mathsf{tree}(x) \ast \mathsf{tree}(y)$

Of course, we would have to prove (in the metatheory) that the least definition exists, but that is not difficult.

2.2 List Segments

We will work with linked lists that use field n for the next element. The predicate for linked list segments is the least satisfying the following specification:

$$\mathsf{ls}(E, F) \iff (E{=}F \wedge \mathsf{emp}) \vee (E{\neq}F \wedge \exists y. E{\mapsto}[n\colon y] \ast \mathsf{ls}(y, F))$$

Once again, this definition allows for additional fields, such as a head field, but the ls predicate is insensitive to the values of these other fields.

With this definition a complete linked list is one that satisfies $\mathsf{ls}(E, \mathsf{nil})$. Complete linked lists, or trees for that matter, are much simpler than segments. But the segments are sometimes needed when reasoning in the middle of a list, particularly for iterative programs. (Similar remarks would apply to iterative tree programs.)

2.3 Examples

For some context and a feel for the sorts of properties expressible, we present a few example procedures with specifications in the fragment in Table 1. We do not discuss loops and procedures in the technical part of the paper, but the techniques we present are strong enough to verify these programs and are used in Smalllfoot to do so.

The disp_tree(p) example accepts any heap in which the argument points to a tree and deallocates the tree, returning the empty heap. As discussed earlier, proving this requires inferring frame axioms at the recursive call sites. Also, this example demonstrates the ability to specify absence of memory leaks, since, if the `dispose p;` command was omitted, then the specification would not hold.

While $\ast$ is required in the proof of disp_tree, it does not appear in the specification. The second example illustrates the use of $\ast$ in specifications, where copy_tree guarantees that after copying a tree, the input tree and the new copy occupy disjoint memory cells.

The third example is the source of the entailment requiring induction discussed in the introduction. This procedure also illustrates how list segments are

Table 1. Example Programs

```
disp_tree(;p)
[tree(p)] {
  local i,j;
  if (p==nil) {}
  else {
    i = p->l;
    j = p->r;
    disp_tree(;i);
    disp_tree(;j);
    dispose p; }
} [emp]
```

```
copy_tree(q;p)
[tree(p)] {
  local i,j,i',j';
  if (p==nil) {}
  else {
    i = p->l;
    j = p->r;
    copy_tree(i';i);
    copy_tree(j';j);
    q = cons();
    q->l = i';
    q->r = j'; }
} [tree(q) ∗ tree(p)]
```

```
append_list(x;y)
[ls(x, nil) ∗ ls(y, nil)] {
  local t,u;
  if (x==nil) {
    x = y; }
  else {
    t = x; u = t->n;
    while (u!=nil)
    [ls(x, t) ∗ t↦[n: u] ∗ ls(u, nil)]
    { t = u;
      u = t->n; }
    t->n = y; }
} [ls(x, nil)]
```

Note that in these examples, assertions are enclosed in square brackets, and procedure parameter lists consist first of the reference parameters, followed by a semicolon, and finally the value parameters.

sometimes needed in loop invariants of code whose specifications only involve complete lists ending in nil.

3 Symbolic Execution

In this section we give rules for triples of the form

$$\{\Pi \mid \Sigma\}\ C\ \{\Pi' \mid \Sigma'\}$$

where C is a loop-free program. The commands C are given by the grammar:

$$\begin{aligned} C ::= {} & \texttt{empty} \mid x{:=}E\ ;\ C \mid x{:=}E{\rightarrow}f\ ;\ C \mid E{\rightarrow}f{:=}F\ ;\ C \\ \mid {} & \texttt{new}(x)\ ;\ C \mid \texttt{dispose}(E)\ ;\ C \mid \texttt{if}\ P\ \texttt{then}\ C\ \texttt{else}\ C\ \texttt{fi}\ ;\ C \end{aligned}$$

The rules in this section appeal to entailments $\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'$ between symbolic heaps. Semantically, entailment is defined by:

$$\Pi \mid \Sigma \vdash \Pi' \mid \Sigma' \text{ is true} \quad \text{iff} \quad \forall s, h.\, s, h \vDash \Pi \mid \Sigma \text{ implies } s, h \vDash \Pi' \mid \Sigma'$$

For the presentation of rules in this section we will regard semantic entailment as an oracle. Soundness of symbolic execution just requires an approximation.

3.1 Operational Rules

The operational rules use the following notation for record expressions:

$$\mathit{mutate}(\rho, f, F) = \begin{cases} f\colon F, \rho' & \text{if } \rho = f\colon E, \rho' \\ f\colon F, \rho & \text{if } f \notin \rho \end{cases} \qquad \mathit{lookup}(\rho, f) = \begin{cases} E & \text{if } \rho = f\colon E, \rho' \\ x \text{ fresh} & \text{if } f \notin \rho \end{cases}$$

The fresh variable returned in the lookup case corresponds to the idea that if a record expression does not give a value for a particular field then we do not

Table 2. Operational Symbolic Execution Rules

Empty

$$\frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\{\Pi \mid \Sigma\}\ \texttt{empty}\ \{\Pi' \mid \Sigma'\}}$$

Assign

$$\frac{\{x{=}E[x'/x] \wedge (\Pi \mid \Sigma)[x'/x]\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma\}\ x{:=}E\ ;\ C\ \{\Pi' \mid \Sigma'\}}\ x' \text{ fresh}$$

Lookup

$$\frac{\{x{=}F[x'/x] \wedge (\Pi \mid \Sigma * E{\mapsto}[\rho])[x'/x]\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * E{\mapsto}[\rho]\}\ x{:=}E{\to}f\ ;\ C\ \{\Pi' \mid \Sigma'\}}\ x' \text{ fresh}, \mathit{lookup}(\rho, f) = F$$

Mutate

$$\frac{\{\Pi \mid \Sigma * E{\mapsto}[\rho']\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * E{\mapsto}[\rho]\}\ E{\to}f{:=}F\ ;\ C\ \{\Pi' \mid \Sigma'\}}\ \mathit{mutate}(\rho, f, F) = \rho'$$

New

$$\frac{\{(\Pi \mid \Sigma)[x'/x] * x{\mapsto}[]\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma\}\ \texttt{new}(x)\ ;\ C\ \{\Pi' \mid \Sigma'\}}\ x' \text{ fresh}$$

Dispose

$$\frac{\{\Pi \mid \Sigma\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * E{\mapsto}[\rho]\}\ \texttt{dispose}(E)\ ;\ C\ \{\Pi' \mid \Sigma'\}}$$

Conditional

$$\frac{\{\Pi \wedge P \mid \Sigma\}\ C_1\ ; C\ \{\Pi' \mid \Sigma'\} \qquad \{\Pi \wedge \neg P \mid \Sigma\}\ C_2\ ; C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma\}\ \texttt{if}\ P\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \texttt{fi}\ ;\ C\ \{\Pi' \mid \Sigma'\}}$$

care what it is. These definitions do not result in conditionals being inserted into record expressions; they do not depend on the values of variables or the heap.

The operational rules are shown in Table 2. One way to understand these rules is by appeal to operational intuition. For instance, reading bottom-up, from conclusion to premise, the Mutate rule says: To determine if $\{\Pi \mid \Sigma * E{\mapsto}[\rho]\}\ E{\to}f{:=}F\ ;\ C\ \{\Pi' \mid \Sigma'\}$ holds, execute $E{\to}f{:=}F$ on the symbolic pre-state $\Pi \mid \Sigma * E{\mapsto}[\rho]$, updating E in place, and then continue with C. Likewise, the Dispose rule says to dispose a symbolic cell (a $\mapsto$ fact), the New rule says to allocate, and the Lookup rule to read. The substitutions of fresh variables are used to keep track of (facts about) previous values of variables.

The role of fresh variables can be understood in terms of standard considerations on Floyd-Hoare logic. Recall that in Floyd's assignment axiom

$$\{A\}\ x{:=}E\ \{\exists x'.\, x{=}E[x'/x] \wedge A[x'/x]\}$$

the fresh variable x' is used to record (at least the existence of) a previous value for x. Our fragment here is quantifier-free, but we can still use the same general idea as in the Floyd axiom, as long as we have an overall postcondition and a continuation of the assignment command.

$$\frac{\{x{=}E[x'/x] \wedge A[x'/x]\}\ C\ \{B\}}{\{A\}\ x{:=}E\ ;\ C\ \{B\}}\ x' \text{ fresh}$$

This rule works in standard Hoare logic: the fact that the Floyd axiom expresses the strongest postcondition translates into its soundness and completeness. All of the rules mentioning fresh variables are obtained in this way from axioms of Separation Logic. This (standard) trick allows use of a quantifier-free language.

We will not explicitly give the semantics of commands, but assume Separation Logic's "fault-avoiding" semantics of triples (as in, e.g., [12]) in:

Theorem 1. *All of the operational rules are sound (preserving validity), and all except for* Dispose *are complete (preserving invalidity).*

To see the incompleteness of the Dispose rule consider:

$$\{x{\mapsto}[] * y{\mapsto}[]\}\ \texttt{dispose}(x)\ ;\ \texttt{empty}\ \{x{\neq}y \mid y{\mapsto}[]\}$$

This is a true triple, but if we apply the Dispose and Empty rules upwards we will be left with an entailment $y{\mapsto}[] \vdash x{\neq}y \mid y{\mapsto}[]$ that is false. The rule loses the implied information that x and y are unequal from the precondition. Although we can construct artificial examples like this that fool the rule, none of the naturally-occurring examples that we have tried in Smallfoot have suffered from it. The reason, so it seems, is that required inequalities tend to be indicated in boolean conditions in programs, in either while loops or conditionals. We have considered hack solutions to this problem but nothing elegant has arisen; so in lieu of practical problems with the incompleteness, we have opted for the simple solution presented here.

This incompleteness could be dealt with if we instead used the backwards-running weakest preconditions of Separation Logic [4]. Unfortunately, there is no existing automatic theorem prover which can deal with the form of these assertions (which use quantification and the separating implication $-\!\!*$). If there were such a prover, we would be eager consumers of it.

3.2 Rearrangement Rules

The operational rules are not sufficient on their own, because some of them expect their preconditions to be in particular forms. For instance, in

$$\{x{=}y \mid z{\mapsto}[f\colon w] * y{\mapsto}[f\colon z]\}\ x{\to}f{:=}y\ ;\ C\ \{\Pi' \mid \Sigma'\}$$

the Mutate rule cannot fire (be applied upwards), because the precondition has to explicitly have $x{\mapsto}[\rho]$ for some ρ.

Symbolic execution has a separate rearrangement phase, which attempts to put the precondition in the proper form for an operational rule to fire. For instance, in the example just given we can observe that the precondition $x{=}y \mid z{\mapsto}[f\colon w] * y{\mapsto}[f\colon z]$ is equivalent to $x{=}y \mid z{\mapsto}[f\colon w] * x{\mapsto}[f\colon z]$, which is in a form that allows the Mutate rule to fire.

We use notation for atomic commands that access heap cell E:

$$A(E) ::= E{\to}f{:=}F \mid x{:=}E{\to}f \mid \texttt{dispose}(E)$$

The basic rearrangement rule simply makes use of equalities to recognize that a dereferencing step is possible.

$$\textsc{Switch}(E)\quad \frac{\{\Pi \mid \Sigma * E{\mapsto}[\rho]\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * F{\mapsto}[\rho]\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}}\ \Pi \mid \Sigma * F{\mapsto}[\rho] \vdash E{=}F$$

For trees and list segments we have rules that expose $\mapsto$ facts by unrolling their inductive definitions, when we have enough information to conclude that the tree or the list is nonempty.[2] A nonempty tree is one that is not nil.

$$\textsc{Unroll Tree}(E)\quad \frac{\{\Pi \mid \Sigma * E{\mapsto}[l{:}\,x', r{:}\,y'] * \mathsf{tree}(x') * \mathsf{tree}(y')\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * \mathsf{tree}(F)\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}}\ \dagger$$

$$^\dagger\text{when } \Pi \mid \Sigma * \mathsf{tree}(F) \vdash F{\neq}\mathsf{nil} \wedge F{=}E \text{ and } x', y' \text{ fresh}$$

Here, we have placed the "side condition", which is necessary for the rule to apply, below it, for space reasons. Besides unrolling the tree definition some matching is included using the equality $F{=}E$.

To unroll a list segment we need to know that the beginning and ending points are different, which implies that it is nonempty.

$$\textsc{Unroll List Segment}(E)\quad \frac{\{\Pi \mid \Sigma * E{\mapsto}[n{:}\,x'] * \mathsf{ls}(x', F')\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * \mathsf{ls}(F, F')\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}}\ \dagger$$

$$^\dagger\text{when } \Pi \mid \Sigma * \mathsf{ls}(F, F') \vdash F{\neq}F' \wedge E{=}F \text{ and } x' \text{ fresh}$$

These rearrangement rules are very deterministic, and are not complete on their own. The reason is that it is possible for an assertion to imply that a cell is allocated, without knowing which $*$-conjunct it necessarily lies in. For example, the assertion $y{\neq}z \mid \mathsf{ls}(x, y) * \mathsf{ls}(x, z)$ contains a "spooky disjunction": it implies that one of the two list segments is nonempty, so that $x{\neq}y \vee x{\neq}z$, but we do not know which. To deal with this in the rearrangement phase we rely on a procedure for exorcising these spooky disjunctions. In essence, $\mathit{exor}(\Pi \mid \Sigma, E)$ is a collection of assertions obtained by doing enough case analysis (adding equalities and inequalities to Π) so that the location of E within a $*$-conjunct is determined. This makes the rearrangement rules complete.

We omit a formal definition of *exor* for space reasons. It is mentioned in the symbolic execution algorithm below, where $\mathit{exor}(g, E)$ is obtained from triple g by applying *exor* to the precondition.

3.3 Symbolic Execution Algorithm

The symbolic execution algorithm works by proof-search using the operational and rearrangement rules. Rearrangement is controlled to ensure termination.

To describe symbolic execution we presume an oracle $\mathit{oracle}(\Pi \mid \Sigma \vdash \Pi' \mid \Sigma')$ for deciding entailments. We also use that we can express consistency of a symbolic heap, and allocatedness of an expression, using entailments:

[2] This is somewhat akin to the "focus" step in shape analysis [14].

$$\begin{aligned} incon(\Pi \mid \Sigma) &\overset{\text{def}}{=} oracle(\Pi \mid \Sigma \vdash \mathsf{nil}{\neq}\mathsf{nil} \mid \mathsf{emp}) \\ allocd(\Pi \mid \Sigma, E) &\overset{\text{def}}{=} incon(\Pi \mid \Sigma * E{\mapsto}[]) \text{ and } incon(E{=}\mathsf{nil} \wedge \Pi \mid \Sigma) \end{aligned}$$

We also use $pre(g)$ to denote the precondition in a Hoare triple g. *incon* and *pre* are used to check the precondition for inconsistency in the first step of the symbolic execution algorithm and *allocd* is used in the second-last line.

Definition 2. *E is active in g if g is of the form*

$$\{\Pi \mid \Sigma\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}$$

Algorithm 3 (Symbolic Execution). *Given a triple g, determines whether or not it is provable.*

```
check(g) =
  if $incon(pre(g))$ return "true"
  if $g$ matches the conclusion of an operational rule
    let $p$ be the premise, or $p_1, p_2$ the two premises in
    if rule EMPTY return $oracle(p)$
    if rule ASSIGN, MUTATE, NEW, DISPOSE, or LOOKUP return $check(p)$
    if rule CONDITIONAL return $check(p_1) \wedge check(p_2)$
  elseif $g$ begins with $A(E)$
    if SWITCH($E$), UNROLL LIST SEGMENT($E$), or UNROLL TREE($E$) applies
      let $p$ be the premise in return $check(p)$
    elseif $allocd(pre(g), E)$ return $\bigwedge\{check(g') \mid g' \in exor(g, E)\}$
    else return "false"
```

Theorem 4. *The Symbolic Execution algorithm terminates, and returns "true" iff there is a proof of the input judgment using the operational and rearrangement rules, where we view each use of an entailment in the symbolic execution rules as a call to the oracle.*

4 Proof Rules for Entailments

The entailment $\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'$ was treated as an oracle in the description of symbolic execution. We now describe a proof theory for entailment.

The rules come in two groups. The first, the normalization rules, get rid of equalities as soon as possible so that the forthcoming rules can be formulated using simple pattern matching (*i.e.*, we can use $E{\mapsto}F$ rather than $E'{\mapsto}F$ plus $E'{=}E$ derivable), make derivable inequalities explicit, perform case analysis using a form of excluded middle, and recognize inconsistency. The second group of rules, the subtraction rules, work by explicating and then removing facts from the right-hand side of an entailment, with the eventual aim of reducing to the axiom $\Pi \mid \mathsf{emp} \vdash \mathsf{true} \mid \mathsf{emp}$.

Before giving the rules, we introduce some notation. We write $op(E)$ as an abbreviation for $E{\mapsto}[\rho]$, $\mathsf{ls}(E, E')$, or $\mathsf{tree}(E)$. The guard $G(op(E))$ is defined by:

$$G(E{\mapsto}[\rho]) \overset{\text{def}}{=} \mathsf{true} \qquad G(\mathsf{ls}(E, E')) \overset{\text{def}}{=} E{\neq}E' \qquad G(\mathsf{tree}(E)) \overset{\text{def}}{=} E{\neq}\mathsf{nil}$$

Table 3. Proof System for Entailment

NORMALIZATION RULES:

$$\frac{}{\Pi \wedge E{\neq}E \mid \Sigma \vdash \Pi' \mid \Sigma'}$$

$$\frac{\Pi[E/x] \mid \Sigma[E/x] \vdash \Pi'[E/x] \mid \Sigma'[E/x]}{\Pi \wedge x{=}E \mid \Sigma \vdash \Pi' \mid \Sigma'} \qquad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge E{=}E \mid \Sigma \vdash \Pi' \mid \Sigma'}$$

$$\frac{\Pi \wedge G(op(E)) \wedge E{\neq}\mathsf{nil} \mid op(E) * \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge G(op(E)) \mid op(E) * \Sigma \vdash \Pi' \mid \Sigma'} \; E{\neq}\mathsf{nil} \notin \Pi \wedge G(op(E))$$

$$\frac{\Pi \wedge E_1{\neq}E_2 \mid op_1(E_1) * op_2(E_2) * \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid op_1(E_1) * op_2(E_2) * \Sigma \vdash \Pi' \mid \Sigma'} \; \begin{array}{l} G(op_1(E_1)), G(op_2(E_2)) \in \Pi \\ E_1{\neq}E_2 \notin \Pi \end{array}$$

$$\frac{\begin{array}{c}\Pi \wedge E_1{=}E_2 \mid \Sigma \vdash \Pi' \mid \Sigma' \\ \Pi \wedge E_1{\neq}E_2 \mid \Sigma \vdash \Pi' \mid \Sigma'\end{array}}{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'} \; \begin{array}{l} E_1 \not\equiv E_2 \\ E_1{=}E_2, E_1{\neq}E_2 \notin \Pi \\ fv(E_1,E_2) \subseteq fv(\Pi,\Sigma,\Pi',\Sigma') \end{array}$$

$$\frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma * \mathsf{tree}(\mathsf{nil}) \vdash \Pi' \mid \Sigma'} \qquad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma * \mathsf{ls}(E,E) \vdash \Pi' \mid \Sigma'}$$

SUBTRACTION RULES:

$$\frac{}{\Pi \mid \mathsf{emp} \vdash \mathsf{true} \mid \mathsf{emp}} \qquad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \wedge E{=}E \mid \Sigma'} \qquad \frac{\Pi \wedge P \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge P \mid \Sigma \vdash \Pi' \wedge P \mid \Sigma'}$$

$$\frac{S \vdash S' \qquad \Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid S * \Sigma \vdash \Pi' \mid S' * \Sigma'} \qquad \frac{}{S \vdash S} \qquad \frac{}{E{\mapsto}[\rho,\rho'] \vdash E{\mapsto}[\rho]}$$

$$\frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \mid \mathsf{tree}(\mathsf{nil}) * \Sigma'} \qquad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \mid \mathsf{ls}(E,E) * \Sigma'}$$

$$\frac{\Pi \mid E{\mapsto}[l\colon E_1, r\colon E_2, \rho] * \Sigma \vdash \Pi' \mid E{\mapsto}[l\colon E_1, r\colon E_2, \rho] * \mathsf{tree}(E_1) * \mathsf{tree}(E_2) * \Sigma'}{\Pi \mid E{\mapsto}[l\colon E_1, r\colon E_2, \rho] * \Sigma \vdash \Pi' \mid \mathsf{tree}(E) * \Sigma'} \; \dagger$$

$$^{\dagger} E{\mapsto}[l\colon E_1, r\colon E_2, \rho] \notin \Sigma'$$

$$\frac{\Pi \wedge E_1{\neq}E_3 \mid E_1{\mapsto}[n\colon E_2, \rho] * \Sigma \vdash \Pi' \mid E_1{\mapsto}[n\colon E_2, \rho] * \mathsf{ls}(E_2, E_3) * \Sigma'}{\Pi \wedge E_1{\neq}E_3 \mid E_1{\mapsto}[n\colon E_2, \rho] * \Sigma \vdash \Pi' \mid \mathsf{ls}(E_1, E_3) * \Sigma'} \; \dagger$$

$$^{\dagger} E_1{\mapsto}[n\colon E_2, \rho] \notin \Sigma'$$

$$\frac{\Pi \mid \mathsf{ls}(E_1, E_2) * \Sigma \vdash \Pi' \mid \mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, \mathsf{nil}) * \Sigma'}{\Pi \mid \mathsf{ls}(E_1, E_2) * \Sigma \vdash \Pi' \mid \mathsf{ls}(E_1, \mathsf{nil}) * \Sigma'}$$

$$\frac{\Pi \wedge G(op(E_3)) \mid \mathsf{ls}(E_1, E_2) * op(E_3) * \Sigma \vdash \Pi' \mid \mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, E_3) * \Sigma'}{\Pi \wedge G(op(E_3)) \mid \mathsf{ls}(E_1, E_2) * op(E_3) * \Sigma \vdash \Pi' \mid \mathsf{ls}(E_1, E_3) * \Sigma'}$$

The proof rules are given in Table 3. Except for $G(op_1(E_1)), G(op_2(E_2)) \in \Pi$, the side-conditions are not needed for soundness, but ensure termination.

Theorem 5 (Soundness and Completeness). *Any provable entailment is valid, and any valid entailment is provable.*

The side-conditions are sufficient to ensure that progress is made when applying rules upwards. Decidability then follows using the naive proof procedure which tries all possibilities, backtracking when necessary.

Theorem 6 (Decidability). *Entailment is decidable.*

It is possible, however, to do much better than the naive procedure. For example, one narrowing of the search space is a phase distinction between normalization and subtraction rules: Any subtraction rule can be commuted above any normalization rule. Further commutations are possible for special classes of assertion, and these are used in Smallfoot.

This system's proof rules can be viewed as coming from certain implications, and are arranged as rules just to avoid the explicit use of the cut rule in proof search. For instance, the fourth normalization rule comes from the implications:

$$E{\mapsto}[] \to E{\neq}\mathsf{nil} \qquad\qquad E_1{\neq}E_2 \wedge \mathsf{ls}(E_1,E_2) \to E_1{\neq}\mathsf{nil}$$

the fifth from the implications:

$$E_1{\mapsto}[\rho_1] * E_2{\mapsto}[\rho_2] \to E_1{\neq}E_2 \qquad E_2{\neq}\mathsf{nil} \wedge E_1{\mapsto}[\rho] * \mathsf{tree}(E_2) \to E_1{\neq}E_2$$

$$E_2{\neq}E_3 \wedge E_1{\mapsto}[\rho] * \mathsf{ls}(E_2,E_3) \to E_1{\neq}E_2$$

$$E_1{\neq}\mathsf{nil} \wedge E_2{\neq}\mathsf{nil} \wedge \mathsf{tree}(E_1) * \mathsf{tree}(E_2) \to E_1{\neq}E_2$$

$$E_1{\neq}\mathsf{nil} \wedge E_2{\neq}E_3 \wedge \mathsf{tree}(E_1) * \mathsf{ls}(E_2,E_3) \to E_1{\neq}E_2$$

$$E_1{\neq}E_3 \wedge E_2{\neq}E_4 \wedge \mathsf{ls}(E_1,E_3) * \mathsf{ls}(E_2,E_4) \to E_1{\neq}E_2$$

and the last two from the implications:

$$\mathsf{tree}(\mathsf{nil}) \to \mathsf{emp} \qquad\qquad \mathsf{ls}(E,E) \to \mathsf{emp}$$

For the inductive predicates, these implications are consequences of unrolling the inductive definition in the metatheory. But note that we do not unroll predicates, instead case analysis via excluded middle takes one judgment to several.

Likewise, the subtraction rules for the inductive predicates are obtained from the implications:

$$\mathsf{emp} \to \mathsf{tree}(\mathsf{nil}) \qquad E{\mapsto}[l{:}\,E_1, r{:}\,E_2, \rho] * \mathsf{tree}(E_1) * \mathsf{tree}(E_2) \to \mathsf{tree}(E)$$

$$\mathsf{emp} \to \mathsf{ls}(E,E) \qquad E_1{\neq}E_3 \wedge E_1{\mapsto}[n{:}\,E_2, \rho] * \mathsf{ls}(E_2,E_3) \to \mathsf{ls}(E_1,E_3)$$

$$\mathsf{ls}(E_1,E_2) * \mathsf{ls}(E_2,\mathsf{nil}) \to \mathsf{ls}(E_1,\mathsf{nil})$$

$$\mathsf{ls}(E_1,E_2) * \mathsf{ls}(E_2,E_3) * E_3{\mapsto}[\rho] \to \mathsf{ls}(E_1,E_3) * E_3{\mapsto}[\rho]$$

$$E_3 \neq \mathsf{nil} \wedge \mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, E_3) * \mathsf{tree}(E_3) \rightarrow \mathsf{ls}(E_1, E_3) * \mathsf{tree}(E_3)$$

$$E_3 \neq E_4 \wedge \mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, E_3) * \mathsf{ls}(E_3, E_4) \rightarrow \mathsf{ls}(E_1, E_3) * \mathsf{ls}(E_3, E_4)$$

The first four are straightforward, while the last four express properties whose verification of soundness would use inductive proofs in the metatheory. The resulting rules do not, however, require a search for inductive premises. In essence, what we generally do is, for each considered inductive predicate, add a collection of rules that are consequences of induction, but that can be formulated in a way that preserves the proof theory's terminating nature.

In the last subtraction rule, the $G(op(E_3)) \wedge op(E_3)$ part of the left-hand side ensures that E_3 does not occur within the segments from E_1 to E_2 or from E_2 to E_3. This is necessary for appending list segments, since they are required to be acyclic.

Here is an example proof, of the entailment mentioned in the Introduction:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{t \neq \mathsf{nil} \mid \mathsf{emp} \vdash \mathsf{emp}}{t \neq \mathsf{nil} \mid \mathsf{ls}(y, \mathsf{nil}) \vdash \mathsf{ls}(y, \mathsf{nil})}}{t \neq \mathsf{nil} \mid t \mapsto [n\colon y] * \mathsf{ls}(y, \mathsf{nil}) \vdash t \mapsto [n\colon y] * \mathsf{ls}(y, \mathsf{nil})}}{t \neq \mathsf{nil} \mid t \mapsto [n\colon y] * \mathsf{ls}(y, \mathsf{nil}) \vdash \mathsf{ls}(t, \mathsf{nil})}}{t \neq \mathsf{nil} \mid \mathsf{ls}(x, t) * t \mapsto [n\colon y] * \mathsf{ls}(y, \mathsf{nil}) \vdash \mathsf{ls}(x, \mathsf{nil})}}{\mathsf{ls}(x, t) * t \mapsto [n\colon y] * \mathsf{ls}(y, \mathsf{nil}) \vdash \mathsf{ls}(x, \mathsf{nil})}$$

Going upwards, this applies the normalization rule which introduces $t \neq \mathsf{nil}$, then the subtraction rule for nil-terminated list segments, the subtraction rule for nonempty list segments, and finally $*$-Introduction (the basic subtraction rule for $*$, which appears fourth) twice.

5 Incomplete Proofs and Frame Axioms

Typically, at a call site to a procedure the symbolic heap will be larger than that required by the procedure's precondition. This is the case in the disp_tree example where, for example, the symbolic heap at one of the recursive call sites is $p \mapsto [l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)$, where that expected by the procedure specification of disp_tree(i) is just $\mathsf{tree}(i)$. We show how to use the proof theory from the previous section to infer frame axioms.

In more detail the (spatial) part of the problem is,

Given: two symbolic heaps, $\Pi \mid \Sigma$ (the heap at the call site), and $\Pi_1 \mid \Sigma_1$ (the procedure precondition)
To Find: a spatial predicate Σ_F, the "spatial frame axiom", satisfying the entailment $\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1 * \Sigma_F$.

Our strategy is to search for a proof of the judgment $\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1$, and if this search, going upwards, halts at $\Pi' \mid \Sigma_F \vdash \mathsf{true} \mid \mathsf{emp}$ then Σ_F is a sound choice as a frame axiom. We give a few examples to show how this mechanism works.

First, and most trivially, let us consider the disp_tree example:

Assertion at Call Site	:	$p \mapsto [l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)$
Procedure Precondition	:	$\mathsf{tree}(i)$

Then an instance of $*$-Introduction

$$\frac{p \mapsto [l\colon i, r\colon j] * \mathsf{tree}(j) \vdash \mathsf{emp}}{p \mapsto [l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j) \vdash \mathsf{tree}(i)}$$

immediately furnishes the correct frame axiom: $p \mapsto [l\colon i, r\colon j] * \mathsf{tree}(j)$.

For an example that requires a little bit more logic, consider:

Assertion at Call Site	:	$x \mapsto [] * y \mapsto []$
Procedure Precondition	:	$x \neq y \mid x \mapsto []$

$$\cfrac{\cfrac{\cfrac{x \neq y \mid y \mapsto [] \vdash \mathsf{emp}}{x \neq y \mid y \mapsto [] \vdash x \neq y \mid \mathsf{emp}}}{x \neq y \mid x \mapsto [] * y \mapsto [] \vdash x \neq y \mid x \mapsto []}}{x \mapsto [] * y \mapsto [] \vdash x \neq y \mid x \mapsto []}$$

Here, the inequality $x \neq y$ is added to the left-hand side in the normalization phase, and then it is removed from the right-hand side in the subtraction phase.

On the other hand, consider what happens in a wrong example:

Assertion at Call Site	:	$x \mapsto [] * y \mapsto []$
Procedure Precondition	:	$x = y \mid x \mapsto []$

$$\cfrac{\cfrac{\cfrac{??}{x \neq y \mid y \mapsto [] \vdash x = y \mid \mathsf{emp}}}{x \neq y \mid x \mapsto [] * y \mapsto [] \vdash x = y \mid x \mapsto []}}{x \mapsto [] * y \mapsto [] \vdash x = y \mid x \mapsto []}$$

In this case we get stuck at an earlier point because we cannot remove the equality $x = y$ from the right-hand side in the subtraction phase. To correctly get a frame axiom we have to obtain true in the pure part of the right-hand side; we do not do so in this case, and we rightly do not find a frame axiom.

The proof-theoretic justification for this method is the following.

Theorem 7. *Suppose that we have an incomplete proof (a proof that doesn't use axioms):*

$$\begin{array}{c}[\Pi' \mid \Sigma_F \vdash \mathsf{true} \mid \mathsf{emp}] \\ \vdots \\ \Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1\end{array}$$

Then there is a complete proof (without premises, using an axiomatic rule at the top) of:

$$\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1 * \Sigma_F .$$

This justifies an extension to the symbolic execution algorithm. In brief, we extend the syntax of loop-free triples with a **jsr** instruction

$$C ::= \cdots \mid [\Pi \mid \Sigma]\ \mathbf{jsr}\ [\Pi' \mid \Sigma'] \,; C \quad \text{jump to subroutine}$$

annotated with a precondition and a postcondition. In Smallfoot this is generated when an annotated program is chopped into straightline Hoare triples. The appropriate operational rule is:

$$\frac{\Pi \mid \Sigma \vdash \Pi_1 \wedge \Pi \mid \Sigma_1 * \Sigma_F \qquad \{\Pi_2 \wedge \Pi \mid \Sigma_2 * \Sigma_F\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma\}\ [\Pi_1 \mid \Sigma_1]\ \mathbf{jsr}\ [\Pi_2 \mid \Sigma_2]\,; C\ \{\Pi' \mid \Sigma'\}}$$

When we encounter a **jsr** command during symbolic execution we run the proof theory from the previous section upwards with goal $\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1$. If it terminates with $\Pi' \mid \Sigma_F \vdash \mathsf{true} \mid \mathsf{emp}$ then we tack Σ_F onto the postcondition Σ_2, and we continue execution with C. Else we report an error.

The description here is simplified. Theorem 7 only considers incomplete proofs with single assumptions, but it is possible to generalize the treatment of frame inference to proofs with multiple assumptions (which leads to several frames being checked in symbolic execution). Also, we have only discussed the spatial part of the frame, neglecting modifies clauses for stack variables. A pure frame must also be discovered, but that is comparatively easy.

Finally, this way of inferring frame axioms works, but is incomplete. To see why, for $[x{\mapsto}-]$ **jsr** $[\mathsf{emp}]$ we run into a variant of the same problem discussed before in incompleteness of the `dispose` instruction: it we added a frame $y{\mapsto}-$ then the postcondition would lose the information that $x{\neq}y$. Similar incompleteness arises for larger-scale operations as well, such as disp_tree. Now, the incompleteness is not completely disastrous. When reasoning about recursive calls to disp_tree, never do we need to conclude an inequality between, say, a just-disposed cell in the left subtree and a cell in the right; $*$ gives us the information we need, at the right time, for the proof to go through.

It is an incompleteness, still.

6 Conclusion

The heap poses great problems for modular verification and analysis. For example, PALE is (purposely) unsound in its treatment of frame axioms for procedures [7], the "modular soundness" of ESC is subtle but probably not definitive [6], interprocedural Shape Analysis is just beginning to become modular [13].

We believe that symbolic execution with Separation Logic has some promise in this area. An initial indication is the local way that heap update is treated in symbolic execution: there is no need to traverse an entire heap structure when an update to a single cell is done, as is the case with Shape Analysis [14]. Going beyond this initial point, it will be essential to have a good way of inferring frame axioms. We have sketched one method here, but there are likely to be others, and what we have done is only a start.

There are similarities between this work and the line of work started by Alias Types [16], however there are crucial differences. One of the most significant points is that here we (completely) axiomatize the consequences of induction for our inductive predicates, while the 'coercions' of [16] include only rolling and unrolling of inductive definitions. Relatedly, here we capture semantic entailment between formulæ exactly, as opposed to providing a coarse approximation. Additionally, this enables commands to branch on possibly inductive consequences of heap shape. Another crucial difference is that here we rely on Separation Logic's

Frame Rule for a very strong form of modularity, and infer frame axioms using incomplete proofs, while Alias Types uses second-order quantification (store polymorphism) with manual instantiation. These differences aside, one wonders whether the lines of work stemming from Alias Types and Separation Logic will someday merge; an interesting step along these lines is in [8], and we are investigating uses of bunched typing [11, 9] for similar purposes.

A different way of automating Separation Logic has recently been put forward by Jia and Walker [5]. An interesting part of their system is how classical arithmetic and substructural logic work together. They also provide a decidable fragment based on Linear Logic. There is a gap between the entailment of their proof theory and that of the heap model, because Linear Logic's proof theory is purposely incomplete for the standard additives supported by the model.

To build on this paper's formulation of symbolic heaps, we would particularly like to have a general scheme of inductive definitions rather than using hardwired predicates. (We are not just asking for semantically well-defined recursive predicates, *e.g.*, as developed in [15], but would want a, hopefully terminating, proof theory.) Soundly extending the techniques here to a class of inductive predicates which generalizes those presented is largely straightforward, since the operational symbolic execution rules would be unaffected, the necessary rearrangement rule for unrolling a more general inductive predicate depends on a certain shape of the inductive definitions where unrolling is triggered by inequalities, and the proof system for entailment would remain sound. Maintaining the present degree of completeness, on the other hand, is nontrivial, since the proof system for entailment becomes incomplete, and exorcising spooky disjunctions may become incomplete (that is, modifying *exor* such that the 'if' direction of Theorem 4 holds is (very) hard).

We would like to relax the restriction to quantifier-free assertions. For example, with $\exists y.\, x{\mapsto}[n\colon y] * \mathsf{ls}(y, x)$ we can describe a circular linked list that has at least one element. It may be that a restricted amount of existential quantification is compatible with having a complete and terminating proof theory.

We should admit that consideration of completeness has greatly slowed us down in this work. The main ideas in this paper were present, and implemented in an early version of Smallfoot, over two years ago. But, the third author (perhaps foolishly) then asked the first two: Is your proof theory complete? And if not, please give an undecidability result, thus rendering completeness impossible. Now, we know that completeness is an ideal that will not always be possible to achieve, but the first two authors were eventually able to answer in the affirmative. Although an ideal, we stress that we would not be satisfied with a sound proof theory based (just) on rolling and unrolling definitions. Having a mechanism to provide axioms for consequences of induction when defining inductive predicates is essential. Without such axioms, it is not possible to verify programs that work at the end of a singly-linked list, or at both ends of a doubly-linked list (we have given similar proof rules for both conventional doubly-linked and xor-linked lists).

That being said, our symbolic execution mechanism is incomplete in two other areas, the treatment of disposal and inference of frame axioms. The former is perhaps reparable by hacks, but the latter is more fundamental.

The ideas in this paper would seem to provide a basis for investigating program analysis. The first crucial question is the exact nature of the abstract domain of formulae, which would enable the calculation of invariants by fixed-point approximation. After that, we would like to attack the problem of modular, interprocedural heap analysis, leveraging the strong modularity properties of Separation Logic.

Acknowledgements. We are grateful to the anonymous referees for helpful comments. During Berdine's stay at Carnegie Mellon University during 2003, this research was sponsored in part by National Science Foundation Grant CCR-0204242. All three authors were supported by the EPSRC.

References

[1] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS 2004*, volume 3328 of *LNCS*, pages 97–109. Springer, Dec. 2004.

[2] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: A tool for checking Separation Logic footprint specifications. In preparation, 2005.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252, 1977.

[4] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL '01*, pages 14–26, 2001.

[5] L. Jia and D. Walker. ILC: A foundation for automated reasoning about pointer programs. Draft., Apr. 2005.

[6] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, 2002.

[7] A. Möller and M. I. Schwartzbach. The pointer assertion logic engine. In *PLDI '01*, pages 221–231, 2001.

[8] G. Morrisett, A. J. Ahmed, and M. Fluet. L^3: A linear language with locations. In P. Urzyczyn, editor, *TLCA*, volume 3461 of *LNCS*, pages 293–307, 2005.

[9] P. W. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, 2003.

[10] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19, 2001.

[11] D. J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.

[12] J. C. Reynolds. Separation Logic: A logic for shared mutable data structures. In *LICS 2002*, pages 55–74. IEEE Computer Society, 2002.

[13] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL '05*, pages 296–309, 2005.

[14] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[15] É.-J. Sims. Extending separation logic with fixpoints and postponed substitution. In *AMAST*, volume 3116 of *LNCS*, pages 475–490. Springer, 2004.

[16] D. Walker and J. G. Morrisett. Alias types for recursive data structures. In *Types in Compilation*, volume 2071 of *LNCS*, pages 177–206, 2001.

An Abstract Interpretation Perspective on Linear vs. Branching Time[*]

Francesco Ranzato and Francesco Tapparo

Dipartimento di Matematica Pura ed Applicata,
Università di Padova, Italy

Abstract. It is known that the branching time language ACTL and the linear time language $\forall\mathrm{LTL}$ of universally quantified formulae of LTL have incomparable expressive powers, i.e., $\mathrm{Sem}(\mathrm{ACTL})$ and $\mathrm{Sem}(\forall\mathrm{LTL})$ are incomparable sets. Within a standard abstract interpretation framework, ACTL can be viewed as an abstract interpretation $\mathrm{LTL}^{\forall}$ of LTL where the universal path quantifier $\forall$ abstracts each linear temporal operator of LTL to a corresponding branching state temporal operator of ACTL. In abstract interpretation terms, it turns out that the universal path quantifier abstraction of LTL is incomplete. In this paper we reason on a generic abstraction α over a domain A of a generic linear time language L. This approach induces both a language $\alpha\mathrm{L}$ of α-abstracted formulae of L and an abstract language L^{α} whose operators are the best correct abstractions in A of the linear operators of L. When the abstraction α is complete for the operators in L it turns out that $\alpha\mathrm{L}$ and L^{α} have the same expressive power, so that trace-based model checking of $\alpha\mathrm{L}$ can be reduced with no lack of precision to A-based model checking of L^{α}. This abstract interpretation-based approach allows to compare temporal languages at different levels of abstraction and to view the standard linear vs. branching time comparison as a particular instance.

1 Introduction

The relationship between linear and branching time specification languages to be used in automatic system verification by model checking has been the subject of thorough investigation [2,8,11,12,13,14,19] (see [20] for a survey). In particular, some of these works [2,8,11,13,14] studied the relationship between the expressive power of linear vs. branching time formalisms.

LTL and CTL are the most commonly used languages for, respectively, linear and branching time model checking. ACTL is the fragment of CTL that uses only the universal path quantifier. Given a Kripke structure $\mathcal{K} = (\Sigma, \xrightarrow{R})$, the standard approach for comparing a linear formula $\varphi \in \mathrm{LTL}$ and a branching formula $\psi \in \mathrm{CTL}$ consists in "abstracting" the path semantics $[\![\varphi]\!] = \{\pi \in \mathit{Path}(\mathcal{K}) \mid \pi \models \varphi\}$ to its corresponding universal (or, dually, existential) state semantics $\{s \in \Sigma \mid \forall \pi \in \mathit{Path}(\mathcal{K}).\, (\pi(0) = s) \Rightarrow \pi \models \varphi\}$ and then comparing this set of states with the standard state semantics $[\![\psi]\!] = \{s \in \Sigma \mid s \models \psi\}$ of ψ. As shown by Cousot and Cousot [5], the intuition

[*] This work was partially supported by the FIRB Project "Abstract interpretation and model checking for the verification of embedded systems" and by the COFIN2004 Project "AIDA".

that this actually is a step of abstraction can be precisely formalized within the abstract interpretation framework [3,4]. In fact, Cousot and Cousot [5] show that the universal path quantifier is an abstraction function $\forall : \wp(\mathrm{Trace}(\Sigma)) \to \wp(\Sigma)$ mapping any set T of traces, viz. arbitrary sequences of states, to the set of states $s \in \Sigma$ such that any path in $\mathcal{K}$ that begins in s belongs to T.

The standard approach introduced by Emerson and Halpern [8] for comparing linear and universal branching time languages relies on the above universal branching abstraction $\forall$. If $\mathrm{L} \subseteq \mathrm{LTL}$ is a linear time language and $\mathcal{L} \subseteq \mathrm{ACTL}$ is a branching time language then L and $\mathcal{L}$ can be compared by comparing the sets $\{\forall(\llbracket\varphi\rrbracket) \subseteq \Sigma \mid \varphi \in \mathrm{L}\}$ and $\{\llbracket\psi\rrbracket \subseteq \Sigma \mid \psi \in \mathcal{L}\}$ in $\wp(\wp(\Sigma))$. Thus, the linear time language L is abstracted to a universal branching time language $\forall\mathrm{L}$, denoted by $\mathrm{B(L)}$ in [8]. For example, it is well known that LTL and ACTL are incomparable (cf. [2,8]), where this means that $\forall\mathrm{LTL}$ and ACTL are incomparable in $\wp(\wp(\Sigma))$.

Moreover, if L is a linear time language which is inductively generated by a set of linear operators $f \in Op_{\mathrm{L}}$ (and a set of atomic propositions p), i.e., $\mathrm{L} \ni \varphi ::= p \mid f(\varphi_1, ..., \varphi_n)$, then the universal path quantifier also induces the following universal state language: $\mathrm{L}^{\forall} \ni \psi ::= \forall p \mid \forall f(\psi_1, ..., \psi_n)$, where each linear temporal operator in *Op* is preceded by the universal path quantifier $\forall$. For example, it turns out that $\mathrm{ACTL} = \mathrm{LTL}^{\forall}$. Thus, the comparison between $\forall\mathrm{LTL}$ and ACTL boils down to the comparison between $\forall\mathrm{LTL}$ and $\mathrm{LTL}^{\forall}$. As a consequence of the incomparability of $\forall\mathrm{LTL}$ and $\mathrm{LTL}^{\forall}$ we obtain that the abstraction map $\forall$ is *incomplete* in the abstract interpretation sense [3,9]. In fact, if $\forall$ would be complete for the operators of LTL then we would also have that $\forall\mathrm{LTL} = \mathrm{LTL}^{\forall}$ whereas this is not the case. Cousot and Cousot [5] analyzed the linear operators that cause the incompleteness of $\forall$ and then isolated some inductive fragments $\mathrm{L} \subseteq \mathrm{LTL}$ such that $\forall\mathrm{L} = \mathrm{L}^{\forall}$.

Thus, abstract interpretation allows to cast the linear vs. branching time problem as a particular instance of a more general "linear vs. A time" problem, where $\alpha : \wp(\mathrm{Trace}(\Sigma)) \to A$ is any abstract interpretation of sets of traces to some abstract domain A. For any such abstraction α, a linear language L therefore induces two "A-time" languages: $\alpha\mathrm{L}$ and L^{α}.

In this paper, we study a number of abstractions of sets of traces that are alternative to the above standard universal path quantifier abstraction. We consider abstractions of $\wp(\mathrm{Trace}(\Sigma))$ parameterized by some model M, namely by the set $Path(\mathcal{K})$ of paths in some Kripke structure $\mathcal{K}$. This is more general than considering abstractions of $\wp(Path(\mathcal{K}))$ because we show that $\wp(Path(\mathcal{K}))$ is a complete (both existential and universal) abstract interpretation of $\wp(\mathrm{Trace}(\Sigma))$. Completeness plays a key role in this generalized approach. In fact, it turns out that when α is complete for the linear operators in Op_{L} of some language L then $\alpha\mathrm{L} = \mathrm{L}^{\alpha}$. We first study an abstract domain consisting of *traces of sets of states*, i.e., $\mathrm{Trace}(\wp(\Sigma))$. Here, the trace of sets abstraction $\mathrm{tr} : \wp(\mathrm{Trace}(\Sigma)) \to \mathrm{Trace}(\wp(\Sigma))$ approximates any set T of traces to the sequence of sets of states reached by some trace in T. This is a more precise abstraction than the universal branching abstraction $\forall$. While tr is not complete for all the linear operators of LTL, we show that tr is instead complete for disjunction, next and eventually operators, namely for the fragment $\mathrm{L}(\{\vee, \mathrm{X}, \mathrm{F}\}) \subseteq \mathrm{LTL}$. We then consider a *reachable state* abstraction $\mathrm{rs} : \wp(\mathrm{Trace}(\Sigma)) \to \wp(\Sigma)$, where any set T of traces T is approximated to the set of states reached by some trace in T. This abstraction is incomparable with the universal branching abstraction $\forall$ while it is less precise than the trace of sets abstraction. In this case, we show that rs is not complete for the next operator and it is still complete for the fragment $\mathrm{L}(\{\vee, \mathrm{F}\})$.

This abstract interpretation-based perspective of the linear vs. branching time problem allows us to show that the Emerson and Halpern [8] transform $\mathrm{EH}_\forall$ of a linear time language L to the branching time language $\forall\mathrm{L}$ actually can be viewed as a "higher-order" abstract interpretation. This means that $\mathrm{EH}_\forall$ is an abstraction from trace abstract domains to universal branching state abstract domains. Hence, the Emerson and Halpern transform can be generalized to a higher-order abstract interpretation $\mathcal{A}_\alpha : \mathrm{AbsDom}(\wp(\mathrm{Trace}(\Sigma))) \to \mathrm{AbsDom}(A)$ which is parameterized by any trace abstraction $\alpha : \wp(\mathrm{Trace}(\Sigma)) \to A$, so that $\mathcal{A}_\alpha(\mathrm{L}) = \{\alpha([\![\varphi]\!]) \mid \varphi \in \mathrm{L}\}$. Therefore, this generalized approach allows to compare the expressive power of linear time languages at any level of abstraction A. As an example, we consider the linear time language $\mathrm{L} \ni \varphi ::= p \mid \mathrm{F}\varphi \mid \mathrm{G}\varphi$. We show how this approach can be used to prove that the languages $\forall\mathrm{L}$ and $\mathrm{L}^\forall$ have incomparable expressive powers by comparing them in a higher-order abstract domain of state partitions.

2 Basic Notions

Notation. Let X be any set. When writing a set $S \in \wp(\wp(X))$ we often use a compact form like in $\{1, 12, 123\} \in \wp(\wp(\{1, 2, 3\}))$. We denote by $\neg$ the complement operator w.r.t. some universe set. A poset or complete lattice C w.r.t. a partial ordering $\leq$ is denoted by $C_\leq$ or $\langle C, \leq\rangle$. A function $f : C \to C$ on a complete lattice C is additive when f preserves arbitrary least upper bounds. We denote by $\mathrm{Part}(X)$ the set of partitions of X. $\mathrm{Part}(X)$ is endowed with the following standard partial order $\preccurlyeq$: given $P_1, P_2 \in \mathrm{Part}(X)$, $P_1 \preccurlyeq P_2$ (P_1 refines P_2) iff $\forall B \in P_1.\exists B' \in P_2.B \subseteq B'$. It turns out that $\langle \mathrm{Part}(X), \preccurlyeq\rangle$ is a complete lattice.

Kripke Structures. We consider transition systems (Σ, R) where the transition relation $R \subseteq \Sigma \times \Sigma$ (also denoted by $\xrightarrow{R}$) is total. A Kripke structure $\mathcal{K} = (\Sigma, R, AP, \ell)$ consists of a transition system (Σ, R) together with a set AP of atomic propositions and a labeling function $\ell : \Sigma \to \wp(AP)$. A trace on Σ is any infinite sequence of elements in Σ, that is, any function $\sigma : \mathbb{N} \to \Sigma$. $\mathrm{Trace}(\Sigma)$ denotes the set of traces on Σ. For any $k \in \mathbb{N}$ and $\sigma \in \mathrm{Trace}(\Sigma)$, $\sigma^k \in \mathrm{Trace}(\Sigma)$ denotes the suffix of σ that begins in $\sigma(k)$, i.e., $\sigma^k = \lambda i \in \mathbb{N}.\sigma(i + k)$. A path in a Kripke structure $\mathcal{K}$ (or, more in general, in a transition system) is any trace $\pi \in \mathrm{Trace}(\Sigma)$ such that for any $i \in \mathbb{N}$, $\pi(i) \xrightarrow{R} \pi(i + 1)$. $\mathit{Path}(\mathcal{K})$ denotes the set of paths in $\mathcal{K}$.

Temporal Languages. LTL and ACTL are two well-known temporal specification languages used in model checking. LTL consists of linear (or path) formulae describing properties of a computation path through linear temporal operators. ACTL consists of branching (or state) formulae that describe properties of computation trees because each temporal operator is preceded by the universal path quantifier. We consider formulae in negation-normal form, so that LTL is inductively defined as follows:

$$\mathrm{LTL} \ni \varphi ::= p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathrm{X}\varphi \mid \mathrm{U}(\varphi_1, \varphi_2) \mid \mathrm{V}(\varphi_1, \varphi_2)$$

where p ranges over a set AP of atomic propositions that contains $true$. Given a Kripke structure $\mathcal{K}$, let us recall the standard semantics $[\![\cdot]\!]_{\mathcal{K}} : \mathrm{LTL} \to \wp(Path(\mathcal{K}))$ of LTL:

- $[\![p]\!]_{\mathcal{K}} = \{\pi \in Path(\mathcal{K}) \mid p \in \ell(\pi(0))\}$;
- $[\![\neg p]\!]_{\mathcal{K}} = \mathrm{Path}(\mathcal{K}) \smallsetminus [\![p]\!]_{\mathcal{K}}$;
- $[\![\varphi_1 \wedge/\vee\ \varphi_2]\!]_{\mathcal{K}} = [\![\varphi_1]\!]_{\mathcal{K}} \cap/\cup\ [\![\varphi_2]\!]_{\mathcal{K}}$;
- "Next": $[\![\mathrm{X}\varphi]\!]_{\mathcal{K}} = \{\pi \in Path(\mathcal{K}) \mid \pi^1 \in [\![\varphi]\!]_{\mathcal{K}}\}$;
- "Until": $[\![\mathrm{U}(\varphi_1, \varphi_2)]\!]_{\mathcal{K}} = \{\pi \in Path(\mathcal{K}) \mid \exists k \in \mathbb{N}.\ \pi^k \in [\![\varphi_2]\!]_{\mathcal{K}}$ and $\forall j \in [0, k).\pi^j \in [\![\varphi_1]\!]_{\mathcal{K}}\}$;
- "Release": $[\![\mathrm{V}(\varphi_1, \varphi_2)]\!]_{\mathcal{K}} = \{\pi \in Path(\mathcal{K}) \mid \forall n \in \mathbb{N}.(\forall i \in [0, n).\pi^i \notin [\![\varphi_1]\!]_{\mathcal{K}}) \Rightarrow (\pi^n \in [\![\varphi_2]\!]_{\mathcal{K}})\}$.

"Globally" (G), "eventually" (F) and "weak-until" (W) can be defined as derived operators in LTL as follows: $\mathrm{G}\varphi \stackrel{\text{def}}{=} \mathrm{V}(\mathit{false}, \varphi)$; $\mathrm{F}\varphi \stackrel{\text{def}}{=} \mathrm{U}(\mathit{true}, \varphi)$; $\mathrm{W}(\varphi_1, \varphi_2) \stackrel{\text{def}}{=} \mathrm{G}\varphi_1 \vee \mathrm{U}(\varphi_1, \varphi_2)$. Moreover, "release" can be expressed in terms of "weak-until": $\mathrm{V}(\varphi_1, \varphi_2) = \mathrm{W}(\varphi_2, \varphi_1 \wedge \varphi_2)$. If *Op* is any set of linear operators then we will denote by $\mathrm{L}(\mathit{Op})$ the subset of LTL formulae which are inductively generated by the grammar:

$$\mathrm{L}(\mathit{Op}) \ni \varphi ::= p \mid \mathit{op}(\varphi_1, ..., \varphi_n)$$

where *op* ranges over *Op*. The universal (or, dually, existential) path quantifier provides a state semantics of LTL. For any $\varphi \in \mathrm{LTL}$, $[\![\forall\varphi]\!]_{\mathcal{K}} = \{s \in \Sigma \mid \forall \pi \in Path(\mathcal{K}).\ (\pi(0) = s) \Rightarrow \pi \in [\![\varphi]\!]_{\mathcal{K}}\}$. For any $\mathrm{L} \subseteq \mathrm{LTL}$, $\forall\mathrm{L}$ denotes the set of universally quantified formulae of L, i.e. $\forall\mathrm{L} = \{\forall\varphi \mid \varphi \in \mathrm{L}\}$.

ACTL is defined by the following grammar:

$$\mathrm{ACTL} \ni \varphi ::= p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathrm{AX}\varphi \mid \mathrm{AU}(\varphi_1, \varphi_2) \mid \mathrm{AV}(\varphi_1, \varphi_2)$$

The standard semantics $[\![\cdot]\!]_{\mathcal{K}} : \mathrm{ACTL} \to \wp(\Sigma)$ w.r.t. a Kripke structure $\mathcal{K}$ goes as follows:

- $[\![p]\!]_{\mathcal{K}} = \{s \in \Sigma \mid p \in \ell(s)\}$;
- $[\![\neg p]\!]_{\mathcal{K}} = \Sigma \smallsetminus [\![p]\!]_{\mathcal{K}}$;
- $[\![\varphi_1 \wedge/\vee\ \varphi_2]\!]_{\mathcal{K}} = [\![\varphi_1]\!]_{\mathcal{K}} \cap/\cup\ [\![\varphi_2]\!]_{\mathcal{K}}$;
- $[\![\mathrm{AX}\varphi]\!]_{\mathcal{K}} = \{s \in \Sigma \mid \forall t \in \Sigma.\ (s \xrightarrow{R} t) \Rightarrow t \in [\![\varphi]\!]_{\mathcal{K}}\}$;
- $[\![\mathrm{AU}(\varphi_1, \varphi_2)]\!]_{\mathcal{K}} = \{s \in \Sigma \mid \forall \pi \in \mathrm{Path}(\mathcal{K}).\ (\pi(0) = s) \Rightarrow \exists k \in \mathbb{N}.\ \pi^k \in [\![\varphi_2]\!]_{\mathcal{K}}$ and $\forall j \in [0, k).\pi^j \in [\![\varphi_1]\!]_{\mathcal{K}}\}$;
- $[\![\mathrm{AV}(\varphi_1, \varphi_2)]\!]_{\mathcal{K}} = \{s \in \Sigma \mid \forall \pi \in \mathrm{Path}(\mathcal{K}).\ (\pi(0) = s) \Rightarrow (\forall n \in \mathbb{N}.\ (\forall i \in [0, n).\pi^i \notin [\![\varphi_1]\!]_{\mathcal{K}}) \Rightarrow (\pi^n \in [\![\varphi_2]\!]_{\mathcal{K}}))\}$.

It is well known that LTL, i.e. $\forall$LTL, and ACTL have incomparable expressive powers [2,8,13,20]. For instance, the ACTL formula AFAGp cannot be expressed in $\forall$LTL while the $\forall$LTL formula AFGp cannot be expressed in ACTL.

3 Abstract Interpretation of Inductive Languages

3.1 Abstract Interpretation Basics

In standard abstract interpretation, abstract domains can be equivalently specified either by *Galois connections/insertions* (GCs/GIs) or by (upper) *closure operators* (uco's) [4].

These two approaches are equivalent, modulo isomorphic representations of domain's objects. The closure operator approach has the advantage of being independent from the representation of domain's objects and is therefore appropriate for reasoning on abstract domains independently from their representation. Recall that $\mu : C \to C$ is a uco when μ is monotone, idempotent and extensive (viz. $x \leq \mu(x)$). It is well known that the set $\mathrm{uco}(C)$ of all uco's on C, endowed with the pointwise ordering $\sqsubseteq$, gives rise to the complete lattice $\langle \mathrm{uco}(C), \sqsubseteq \rangle$ of abstract domains of C. The ordering on $\mathrm{uco}(C)$ corresponds to the standard order which is used to compare abstract domains with regard to their precision: $\mu_1 \sqsubseteq \mu_2$ means that the domain μ_1 is a more precise abstraction of C than μ_2, or, equivalently, that the abstract domain μ_1 is a refinement of μ_2. Each closure $\mu \in \mathrm{uco}(C)$ is uniquely determined by the set $\mu(C)$ of its fixpoints, which is also its image $\mathrm{img}(\mu)$. Moreover, a subset $X \subseteq C$ is the set of fixpoints of a uco on C iff X is meet-closed (i.e. closed under arbitrary greatest lower bounds). Also, we have that $\mu \sqsubseteq \rho$ iff $\rho(C) \subseteq \mu(C)$. Often, we will identify closures with their sets of fixpoints since this does not give rise to ambiguity.

We denote by $\mathcal{G} = (\alpha, C, A, \gamma)$ a GC/GI of the abstract domain A into the concrete domain C through the abstraction and concretization maps α and γ forming an adjunction between C and A: $\alpha(c) \leq_C a \Leftrightarrow c \leq_A \gamma(a)$. Let us recall that it is enough to specify either the abstraction or the concretization map because in any GC the left adjoint map α determines the right adjoint map γ and vice versa: on the one hand, α is additive iff α admits the right adjoint $\gamma(a) = \vee_C\{c \in C \mid \alpha(c) \leq_A a\}$; on the other hand, γ is co-additive iff γ admits the left adjoint map $\alpha(c) = \wedge_A\{a \in A \mid c \leq_C \gamma(a)\}$. Recall that a GC is a GI when α is onto (or, equivalently, γ is 1-1), meaning that A does not contain useless abstract values. Recall that any GC $\mathcal{G}$ induces the uco $\mu_{\mathcal{G}} = \gamma \circ \alpha$ and conversely any $\mu \in \mathrm{uco}(C)$ induces a GI $(\mu, C, \mathrm{img}(\mu), \mathrm{id})$. Galois connections of a concrete domain C can be ordered according to their precision by exploiting the above ordering on the induced uco's: $\mathcal{G}_1 = (\alpha_1, C, A_1, \gamma_1) \leq \mathcal{G}_2 = (\alpha_2, C, A_2, \gamma_2)$ when $\mu_{\mathcal{G}_1} \sqsubseteq \mu_{\mathcal{G}_2}$. Let $\alpha : \wp(X) \to \wp(Y)$ and $\gamma : \wp(Y) \to \wp(X)$, and $\widetilde{\alpha} \stackrel{\text{def}}{=} \neg \circ \alpha \circ \neg$ and $\widetilde{\gamma} \stackrel{\text{def}}{=} \neg \circ \gamma \circ \neg$. Recall that $(\alpha, \wp(X)_{\subseteq/\supseteq}, \wp(Y)_{\subseteq/\supseteq}, \gamma)$ is a GC/GI iff $(\widetilde{\alpha}, \wp(X)_{\supseteq/\subseteq}, \wp(Y)_{\supseteq/\subseteq}, \widetilde{\gamma})$ is a GC/GI. Thus, results on α/γ and $\widetilde{\alpha}/\widetilde{\gamma}$ can be dualized through complementation.

By the above equivalence, throughout the paper, $\mathrm{uco}(C)_{\sqsubseteq}$ will play the role of the lattice of abstract interpretations of C [3,4], i.e. the complete lattice of all the abstract domains of the concrete domain C.

Let (α, C, A, γ) be a GI, $f : C \to C$ be some concrete semantic function — for simplicity of notation, we consider here 1-ary functions — and $f^\sharp : A \to A$ be a corresponding abstract semantic function. Then, $\langle A, f^\sharp \rangle$ is a sound abstract interpretation when $\alpha \circ f \sqsubseteq f^\sharp \circ \alpha$. The abstract function $f^A \stackrel{\text{def}}{=} \alpha \circ f \circ \gamma : A \to A$ is called the *best correct approximation* of f in A. *Completeness* in abstract interpretation [3,9] corresponds to require the following strengthening of soundness: $\alpha \circ f = f^\sharp \circ \alpha$. Hence, completeness corresponds to require that, in addition to soundness, no loss of precision is introduced by the abstract function $f^\sharp$ on the approximation $\alpha(c)$ of a concrete object $c \in C$ with respect to approximating by α the concrete computation $f(c)$. Completeness is an abstract domain property because it only depends on the abstract domain: in fact, it turns out that $\langle A, f^\sharp \rangle$ is complete iff $\langle A, f^A \rangle$ is complete. Thus, completeness can be equivalently stated as a property of closures as follows: $\mu \in \mathrm{uco}(C)$ is complete for f iff $\mu \circ f = \mu \circ f \circ \mu$ [9].

3.2 Abstract Semantics of Inductive Languages

Concrete Semantics. It is well known that abstract interpretation can be applied to approximate the semantics of any inductively defined language. Assume that formulae of a generic inductive language L are defined by:

$$\mathrm{L} \ni \varphi ::= p \mid f(\varphi_1, ..., \varphi_n)$$

where p ranges over a set of atomic propositions, that is left unspecified, while f ranges over a finite set Op of operators. Each operator $f \in Op$ has an arity[1] $\sharp(f) > 0$. The set of operators of L is also denoted by Op_{L}. Formulae in L are interpreted on a *semantic structure* $\mathcal{S} = (C, AP, I)$ where: C is any (concrete) domain of interpretation, AP is a set of atomic propositions and I is an interpretation function such that for any $p \in AP$, $I(p) \in C$ and for any $f \in Op$, $I(f) : C^{\sharp(f)} \to C$. For $p \in AP$ and $f \in Op$ we will also use $\boldsymbol{p}$ and $\boldsymbol{f}$ to denote, respectively, $I(p)$ and $I(f)$. Also, $\boldsymbol{Op} \stackrel{\text{def}}{=} \{\boldsymbol{f} \mid f \in Op\}$. Hence, the *concrete semantic function* $[\![\cdot]\!]_{\mathcal{S}} : \mathrm{L} \to C$ is inductively defined as follows:

$$[\![p]\!]_{\mathcal{S}} = \boldsymbol{p} \quad \text{and} \quad [\![f(\varphi_1, ..., \varphi_n)]\!]_{\mathcal{S}} = \boldsymbol{f}([\![\varphi_1]\!]_{\mathcal{S}}, ..., [\![\varphi_n]\!]_{\mathcal{S}}).$$

When clear from the context, we will omit the subscript $\mathcal{S}$ which denotes the underlying semantic structure. The set of semantic evaluations in $\mathcal{S}$ of formulae in L is denoted by $\mathrm{Sem}_{\mathcal{S}}(\mathrm{L}) \stackrel{\text{def}}{=} \{[\![\varphi]\!]_{\mathcal{S}} \mid \varphi \in \mathrm{L}\}$, or simply by $\mathrm{Sem}_C(\mathrm{L})$. $\mathrm{Sem}_C(\mathrm{L})$ is also called the *expressive power* (in C) of the language L.

If g is any syntactic operator with arity $\sharp(g) = n > 0$ and whose interpretation is given by $\boldsymbol{g} : C^n \to C$ then we say that a language L is *closed under* g when for any $\varphi_1, ..., \varphi_n \in \mathrm{L}$ there exists some $\psi \in \mathrm{L}$ such that $\boldsymbol{g}([\![\varphi_1]\!]_{\mathcal{S}}, ..., [\![\varphi_n]\!]_{\mathcal{S}}) = [\![\psi]\!]_{\mathcal{S}}$, for any semantic structure $\mathcal{S}$. In particular, if L is evaluated on a powerset $\wp(X)$ then L is closed under (infinite) logical conjunction iff for any $\Phi \subseteq \mathrm{L}$, there exists some $\psi \in \mathrm{L}$ such that $\bigcap_{\varphi \in \Phi} [\![\varphi]\!]_{\mathcal{S}} = [\![\psi]\!]_{\mathcal{S}}$.

The standard semantics of LTL and ACTL, as recalled in Section 2, can be viewed as concrete semantic functions, where the concrete semantic domains are given, respectively, by $\wp(\mathrm{Path}(\mathcal{K}))$ and $\wp(\Sigma)$.

Comparing Expressive Power. The standard notion of expressive power is used to compare different languages. Let L_1 and L_2 be two languages. L_1 is more expressive than L_2, denoted by $\mathrm{L}_1 \leq \mathrm{L}_2$, when for any semantic structure $\mathcal{S} = (C, AP, I)$ such that I provides an interpretation for all the operators in L_1 and L_2, $\mathrm{Sem}_{\mathcal{S}}(\mathrm{L}_2) \subseteq \mathrm{Sem}_{\mathcal{S}}(\mathrm{L}_1)$, while L_1 is equivalent to L_2, denoted by $\mathrm{L}_1 \equiv \mathrm{L}_2$, when $\mathrm{L}_1 \leq \mathrm{L}_2$ and $\mathrm{L}_2 \leq \mathrm{L}_1$, viz., $\mathrm{Sem}_{\mathcal{S}}(\mathrm{L}_1) = \mathrm{Sem}_{\mathcal{S}}(\mathrm{L}_2)$. For instance, as recalled in Section 2, ACTL and $\forall$LTL have incomparable expressive powers meaning that ACTL $\not\leq$ $\forall$LTL and $\forall$LTL $\not\leq$ ACTL.

Abstract Semantics. Within the standard abstract interpretation framework for defining abstract semantics [3,4], for a given semantic structure $\mathcal{S} = (C, AP, I)$, $C_{\leq}$ is a

[1] It would be possible to consider generic operators whose arity is any possibly infinite ordinal, thus allowing, for example, infinite conjunctions or disjunctions.

complete lattice which plays the role of concrete domain. Let us consider an abstract domain A specified by a GI $\mathcal{G} = (\alpha, C, A, \gamma)$. Thus, A induces an abstract semantic structure $\mathcal{S}^A = (A, AP, I^A)$ where I^A is defined through best correct approximations as follows:

$$I^A(p) \stackrel{\text{def}}{=} \alpha(I(p)) \quad \text{and} \quad I^A(f) \stackrel{\text{def}}{=} \alpha \circ I(f) \circ \gamma.$$

Thus, $\mathcal{S}^A$ induces the *abstract semantic function* $[\![\cdot]\!]^A_{\mathcal{S}} : \mathrm{L} \to A$ (also simply denoted by $[\![\cdot]\!]^A$). We will also use L^α or L^A to denote the abstract semantic evaluation of L induced by the abstract domain A so that $\mathrm{Sem}(\mathrm{L}^\alpha)$ (or $\mathrm{Sem}(\mathrm{L}^A)$) denotes the set of abstract semantics $\{[\![\varphi]\!]^A \mid \varphi \in \mathrm{L}\}$.

On the other hand, the domain A also induces the overall abstraction of the concrete semantics, namely ${}^A[\![\cdot]\!]_{\mathcal{S}} : \mathrm{L} \to A$ is defined by: ${}^A[\![\varphi]\!]_{\mathcal{S}} \stackrel{\text{def}}{=} \alpha([\![\varphi]\!]_{\mathcal{S}})$. In this case, we will use $\alpha\mathrm{L}$ to denote this abstract semantic evaluation of L induced by the abstract domain A so that $\mathrm{Sem}(\alpha\mathrm{L}) = \{{}^A[\![\varphi]\!]_{\mathcal{S}} \mid \varphi \in \mathrm{L}\}$.

Definition 1. The abstraction α is *complete* for L when $\mathrm{Sem}(\alpha\mathrm{L}) = \mathrm{Sem}(\mathrm{L}^\alpha)$.

This is indeed a generalization of Emerson and Halpern's [8] approach for comparing linear and branching formulae based on the universal path quantifier $\forall$. In fact, we will see in Section 4.2 how the universal path quantifier $\forall$ can be viewed as a particular abstraction of sets of traces, so that the branching language $\forall\mathrm{L} = \{\forall\varphi \mid \varphi \in \mathrm{L}\}$ and the corresponding results in [8] can be cast as particular cases in our framework.

It turns out that the abstract semantic function is always sound by construction: for any $\varphi \in \mathrm{L}$, $\alpha([\![\varphi]\!]_{\mathcal{S}}) \leq_A [\![\varphi]\!]^A_{\mathcal{S}}$ (or, equivalently, $[\![\varphi]\!]_{\mathcal{S}} \leq_C \gamma([\![\varphi]\!]^A_{\mathcal{S}})$). As far as completeness is concerned, it turns out that completeness of the abstract domain A for (the interpretation of) the operators in *Op* ensures completeness of the abstract semantic function [5].

Theorem 1 (Cousot and Cousot [5]). *If A is complete for every $\boldsymbol{f} \in \boldsymbol{Op}_{\mathrm{L}}$ then for every $\varphi \in \mathrm{L}$, $\alpha([\![\varphi]\!]_{\mathcal{S}}) = [\![\varphi]\!]^A_{\mathcal{S}}$. In this case, α is complete for* L.

4 Abstracting Traces

4.1 Trace Semantics of Linear Languages

As recalled above, the standard semantics of a linear formula $\varphi \in \mathrm{LTL}$ consists of a set of paths in a Kripke structure $\mathcal{K} = (\Sigma, R, AP, \ell)$. $\mathrm{Path}(\mathcal{K})$ can be viewed as a *model* M for interpreting LTL. It turns out that this standard semantics can be obtained as an abstract interpretation of a more general semantics which evaluates formulae in LTL as a set of traces and therefore is independent from a given model. Following [5], this *trace semantics* $[\![\cdot]\!] : \mathrm{LTL} \to \wp(\mathrm{Trace}(\Sigma))$ only depends on a state space Σ and is as follows:

- $[\![p]\!] = \{\sigma \in \mathrm{Trace}(\Sigma) \mid p \in \ell(\sigma(0))\}$;
- $[\![\neg p]\!] = \mathrm{Trace}(\Sigma) \smallsetminus [\![p]\!]$;
- $[\![\varphi_1 \wedge/\vee\ \varphi_2]\!] = [\![\varphi_1]\!] \cap/\cup [\![\varphi_2]\!]$;
- $[\![\mathrm{X}\varphi]\!] = \mathbf{X}([\![\varphi]\!]) \stackrel{\text{def}}{=} \{\sigma \in \mathrm{Trace}(\Sigma) \mid \sigma^1 \in [\![\varphi]\!]\}$;

– $[\![\mathrm{U}(\varphi_1,\varphi_2)]\!] = \mathbf{U}([\![\varphi_1,\varphi_2]\!]) \stackrel{\text{def}}{=} \{\sigma \in \mathrm{Trace}(\Sigma) \mid \exists k \in \mathbb{N}.\ \sigma^k \in [\![\varphi_2]\!]$ and $\forall j \in [0,k).\sigma^j \in [\![\varphi_1]\!]\}$;
– $[\![\mathrm{V}(\varphi_1,\varphi_2)]\!] = \mathbf{V}([\![\varphi_1,\varphi_2]\!]) \stackrel{\text{def}}{=} \{\sigma \in \mathrm{Trace}(\Sigma) \mid \forall n \in \mathbb{N}.\ (\forall i \in [0,n).\sigma^i \notin [\![\varphi_1]\!]) \Rightarrow (\sigma^n \in [\![\varphi_2]\!])\}$.

Let $M = \mathit{Path}(\mathcal{K})$ be a model and let us define $\alpha_{M_\forall} : \wp(\mathrm{Trace}(\Sigma)) \to \wp(M)$ and $\gamma_{M_\forall} : \wp(M) \to \wp(\mathrm{Trace}(\Sigma))$ as follows:

$$\alpha_{M_\forall}(T) \stackrel{\text{def}}{=} T \cap M \qquad \text{and} \qquad \gamma_{M_\forall}(P) \stackrel{\text{def}}{=} P.$$

It is easy to note that $(\alpha_{M_\forall}, \wp(\mathrm{Trace}(\Sigma))_\supseteq, \wp(M)_\supseteq, \gamma_{M_\forall})$ is a GI. This is a *universal model abstraction* (hence the subscript $\forall$) because sets of traces and paths are ordered by superset inclusion. The abstraction map $\lambda T.T \cap M$ on $\wp(\mathrm{Trace}(\Sigma))_\subseteq$ gives also rise to the *existential model abstraction* $(\alpha_{M_\exists}, \wp(\mathrm{Trace}(\Sigma))_\subseteq, \wp(M)_\subseteq, \gamma_{M_\exists})$ where:

$$\alpha_{M_\exists}(T) \stackrel{\text{def}}{=} T \cap M \qquad \text{and} \qquad \gamma_{M_\exists}(P) \stackrel{\text{def}}{=} P \cup \neg M.$$

Note that $\neg M$ is the set of "spurious" traces, namely traces that are not paths. This is a GI as well. Existential abstraction is dual to universal abstraction because: $\gamma_{M_\exists} \circ \alpha_{M_\exists} = \neg \circ (\gamma_{M_\forall} \circ \alpha_{M_\forall}) \circ \neg$.

It is immediate to notice that for any $\varphi \in \mathrm{LTL}$, $[\![\varphi]\!]_{\mathcal{K}} = \alpha_{M_\forall}([\![\varphi]\!]) = \alpha_{M_\exists}([\![\varphi]\!])$. In abstract interpretation terms, this is a consequence of the fact that the standard path semantics of LTL is a complete abstract interpretation of trace semantics.

Proposition 1. *$\alpha_{M_\forall}$ and $\alpha_{M_\exists}$ are complete for the linear operators in $\boldsymbol{Op}_{\mathrm{LTL}}$.*

As a consequence, $\alpha_{M_\forall}([\![\varphi]\!]) = [\![\varphi]\!]^{\alpha_{M_\forall}} = [\![\varphi]\!]_{\mathcal{K}}$ and $\alpha_{M_\exists}([\![\varphi]\!]) = [\![\varphi]\!]^{\alpha_{M_\exists}} = [\![\varphi]\!]_{\mathcal{K}}$, namely path semantics can be retrieved as complete abstractions of trace semantics.

In the following, we provide a number of abstractions of the trace semantics of LTL, based on abstract domains of the existential/universal concrete domain $\wp(\mathrm{Trace}(\Sigma))_{\subseteq/\supseteq}$. Any such trace abstraction $\alpha_M^{\exists/\forall} : \wp(\mathrm{Trace}(\Sigma))_{\subseteq/\supseteq} \to A$ depends on a model $M = \mathit{Path}(\mathcal{K})$ and can be factorized as $\alpha_M^{\exists/\forall} = \alpha^{\exists/\forall} \circ \alpha_{M_{\exists/\forall}}$, where $\alpha^{\exists/\forall} : \wp(M)_{\subseteq/\supseteq} \to A$ is a path abstraction, namely an abstraction of of the existential/universal domain $\wp(M)_{\subseteq/\supseteq}$ of sets of paths. It turns out that the above Proposition 1 makes completeness of trace and path abstractions $\alpha_M^{\exists/\forall}$ and $\alpha^{\exists/\forall}$ equivalent. In fact, if $\boldsymbol{f} : \wp(\mathrm{Trace}(\Sigma)) \to \wp(\mathrm{Trace}(\Sigma))$ is a linear trace operator and $\boldsymbol{f}^{\exists/\forall} : \wp(M) \to \wp(M)$ is the corresponding linear path operator induced by $\alpha_{M_{\exists/\forall}}$ (i.e., $\boldsymbol{f}^{\exists/\forall} = \alpha_{M_{\exists/\forall}} \circ \boldsymbol{f} \circ \gamma_{M_{\exists/\forall}}$), then $\alpha_M^{\exists/\forall}$ is complete for $\boldsymbol{f}$ iff $\alpha^{\exists/\forall}$ is complete for $\boldsymbol{f}^{\exists/\forall}$.

4.2 Branching Abstraction

As shown by Cousot and Cousot [5], the universal path quantifier allows to cast states as an abstraction of traces, so that state-based model checking can be viewed as an abstraction of trace-based model checking. Let $\mathcal{K}$ be a Kripke structure and $M = \mathit{Path}(\mathcal{K})$ be the corresponding model. For any $s \in \Sigma$ and $i \in \mathbb{N}$, we define $M^i_{\downarrow s} \stackrel{\text{def}}{=} \{\pi \in M \mid \pi(i) = s\}$. Therefore, $M^i_{\downarrow s}$ is the set of paths in M whose state at time i is s. In particular, $M^0_{\downarrow s}$ is the set of paths that start in s. The *universal branching abstraction* $\mathcal{G}^\forall = (\alpha^\forall_M, \wp(\mathrm{Trace}(\Sigma))_\supseteq, \wp(\Sigma)_\supseteq, \gamma^\forall_M)$ is defined as follows:

$$\alpha^\forall_M(T) \stackrel{\text{def}}{=} \{s \in \Sigma \mid M^0_{\downarrow s} \subseteq T\} \qquad \text{and} \qquad \gamma^\forall_M(S) \stackrel{\text{def}}{=} \{\pi \in M \mid \pi(0) \in S\}.$$

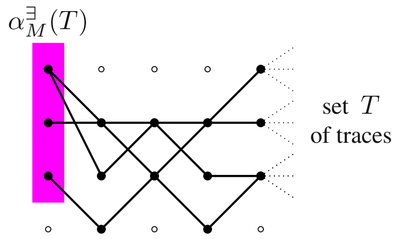


Fig. 1. Branching abstraction

It turns out that $\mathcal{G}^{\forall}$ is a GI. The *existential* branching abstraction is defined by duality:

- $\alpha^{\exists}_{M}(T) \stackrel{\text{def}}{=} \neg(\alpha^{\forall}_{M}(\neg(T))) = \{s \in \Sigma \mid M^{0}_{\downarrow s} \cap T \neq \varnothing\}$;
- $\gamma^{\exists}_{M}(S) \stackrel{\text{def}}{=} \neg(\gamma^{\forall}_{M}(\neg(S))) = \{\pi \in \wp(\text{Trace}(\Sigma)) \mid (\pi \in M) \Rightarrow (\pi(0) \in S)\}$.

In this case, $\mathcal{G}^{\exists} = (\alpha^{\exists}_{M}, \wp(\text{Trace}(\Sigma))_{\subseteq}, \wp(\Sigma)_{\subseteq}, \gamma^{\exists}_{M})$ is a GI. An example of existential branching abstraction is depicted in Figure 1.

The branching abstraction exactly formalizes the universal path quantification of LTL formulae: in fact, it is immediate to observe that for any $\varphi \in$ LTL, $[\![\forall\varphi]\!]_{\mathcal{K}} = \alpha^{\forall}_{M}([\![\varphi]\!]_{\mathcal{K}})$. Moreover, as shown by Cousot and Cousot [5], it turns out that the branching abstraction $\text{LTL}^{\alpha^{\forall}_{M}}$ of LTL exactly gives ACTL, namely the best correct approximations of the linear operators of LTL induced by $\alpha^{\forall}_{M}$ coincide with the branching state temporal operators of ACTL. Therefore, $\text{Sem}(\text{LTL}^{\alpha^{\forall}_{M}}) = \text{Sem}(\text{ACTL})$.

As recalled above, it is well known that $\forall$LTL and ACTL have incomparable expressive powers. In our framework, this means that $\text{Sem}(\alpha^{\forall}_{M}\text{LTL})$ and $\text{Sem}(\text{LTL}^{\alpha^{\forall}_{M}}) = \text{Sem}(\text{ACTL})$ are incomparable sets, i.e. the branching abstraction is incomplete for LTL. As a consequence, by Theorem 1, it turns out that the branching abstraction is incomplete for some operators in $\boldsymbol{Op}_{\text{LTL}}$. The sources of incompleteness of $\alpha^{\forall}_{M}$ have been analyzed by Cousot and Cousot [5]: the branching abstraction results to be incomplete for the disjunction, until and release operators (see [5]). On the other hand, Maidl [14] provides a synctatic characterization of the maximum common fragment, called LTL_{det}, of LTL and ACTL: for any $\varphi \in$ LTL, $\alpha^{\forall}_{M}([\![\varphi]\!]) \in \text{Sem}(\text{ACTL})$ iff $[\![\varphi]\!] \in \text{Sem}(\text{LTL}_{\text{det}})$. We will further discuss completeness of the branching abstraction in Section 5.1.

4.3 Trace of Sets Abstraction

Sets of traces can be approximated by a trace of sets. Let us formalize this approximation. We consider the abstract domain $\text{Trace}(\wp(\Sigma))$, namely sequences of sets of states. Traces of sets are ordered pointwise: if $\sigma, \tau \in \text{Trace}(\wp(\Sigma))$ then $\sigma \sqsubseteq \tau$ iff $\forall i \in \mathbb{N}.\ \sigma(i) \subseteq \tau(i)$. Thus, we first consider existential traces of sets because $\wp(\Sigma)$ is here ordered by $\subseteq$. It turns out that $\text{Trace}(\wp(\Sigma))_{\sqsubseteq}$ is a complete lattice where $\sigma \sqcup \tau = \lambda i.\ \sigma(i) \cup \tau(i)$ and $\sigma \sqcap \tau = \lambda i.\ \sigma(i) \cap \tau(i)$. The *existential trace of sets abstraction* $\alpha^{\exists t}_{M} : \wp(\text{Trace}(\Sigma)) \to \text{Trace}(\wp(\Sigma))$ is then defined as follows:

$$\alpha^{\exists t}_{M}(T) \stackrel{\text{def}}{=} \lambda i \in \mathbb{N}.\{\pi(i) \mid \pi \in T \cap M\} = \lambda i \in \mathbb{N}.\{s \in \Sigma \mid M^{i}_{\downarrow s} \cap T \neq \varnothing\}$$

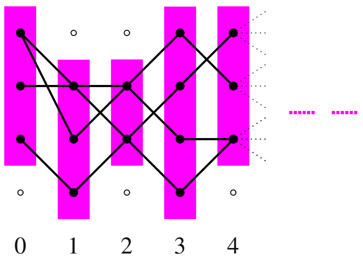


Fig. 2. Existential trace of sets abstraction

$$\begin{tikzcd} a \arrow[r] & b \arrow[loop right] \end{tikzcd} \qquad \begin{tikzcd} a \arrow[r] \arrow[dr] & b \arrow[loop right] \\ c \arrow[r] \arrow[ur] & d \arrow[loop right] \end{tikzcd} \qquad \begin{tikzcd} a \arrow[r, bend left] & b \arrow[l, bend left] \arrow[loop above] & c \arrow[l] \end{tikzcd}$$

Fig. 3. Transition systems $\mathcal{T}_1$ (left), $\mathcal{T}_2$ (middle) and $\mathcal{T}_3$ (right)

and together with its adjoint map $\gamma_M^{\exists t} : \mathrm{Trace}(\wp(\Sigma)) \to \wp(\mathrm{Trace}(\Sigma))$ defined by

$$\gamma_M^{\exists t}(\tau) \stackrel{\text{def}}{=} \{\pi \in \mathrm{Trace}(\Sigma) \mid (\pi \in M) \Rightarrow \forall i \in \mathbb{N}.\ \pi(i) \in \tau(i)\}$$

gives rise to a GC.

Theorem 2. $\mathcal{G}^{\exists t} = (\alpha_M^{\exists t}, \wp(\mathrm{Trace}(\Sigma))_{\subseteq}, \mathrm{Trace}(\wp(\Sigma))_{\sqsubseteq}, \gamma_M^{\exists t})$ *is a GC.*

A graphical example of an existential trace of sets abstraction is given in Figure 2. It turns out that $\mathcal{G}^{\exists t}$ is not a GI. In fact, for the transition system $\mathcal{T}_1$ in Figure 3, $\gamma_M^{\exists t}$ is not 1-1: $\gamma_M^{\exists t}(\langle\{a\},\{a\},\{b\},\{b\},\{b\},...\rangle) = \gamma_M^{\exists t}(\langle\{a\},\{a\},\{a\},\{b\},\{b\},...\rangle) = \neg M$.

The *universal* trace of sets abstraction is dually defined, where the complement of a trace of sets is defined pointwise, namely for any $\tau \in \mathrm{Trace}(\wp(\Sigma))$, $\neg\tau = \lambda i.\ \neg\tau(i)$.

- $\alpha_M^{\forall t}(T) \stackrel{\text{def}}{=} \neg(\alpha_M^{\exists t}(\neg(T))) = \lambda i \in \mathbb{N}.\{s \in \Sigma \mid M_{\downarrow s}^i \subseteq T\}$;
- $\gamma_M^{\forall t}(\tau) \stackrel{\text{def}}{=} \neg(\gamma_M^{\exists t}(\neg\tau)) = \{\pi \in M \mid \exists i \in \mathbb{N}.\pi(i) \in \tau(i)\}$.

Hence, $\mathcal{G}^{\forall t} = (\alpha_M^{\forall t}, \wp(\mathrm{Trace}(\Sigma))_{\supseteq}, \mathrm{Trace}(\wp(\Sigma))_{\sqsupseteq}, \gamma_M^{\forall t})$ is a GC as well.

4.4 Reachable State Abstraction

One can approximate a set T of traces through the set of states that can be reached by some path in T. Let us formalize this approximation. For any $s \in \Sigma$, let us define $M_{\downarrow s} \stackrel{\text{def}}{=} \bigcup_{i \in \mathbb{N}} M_{\downarrow s}^i = \{\pi \in M \mid \exists i \in \mathbb{N}.\ \pi(i) = s\}$. Thus, the *existential reachable state abstraction* $\alpha_M^{\exists r} : \wp(\mathrm{Trace}(\Sigma)) \to \wp(\Sigma)$ is defined by:

$$\alpha_M^{\exists r}(T) \stackrel{\text{def}}{=} \{s \in \Sigma \mid M_{\downarrow s} \cap T \neq \varnothing\}.$$

Therefore, the corresponding concretization map $\gamma_M^{\exists r} : \wp(\Sigma) \to \wp(\mathrm{Trace}(\Sigma))$ is as follows:

$$\gamma_M^{\exists r}(S) \stackrel{\text{def}}{=} \{\pi \in \mathrm{Trace}(\Sigma) \mid (\pi \in M) \Rightarrow (\forall i \in \mathbb{N}.\ \pi(i) \in S)\}.$$

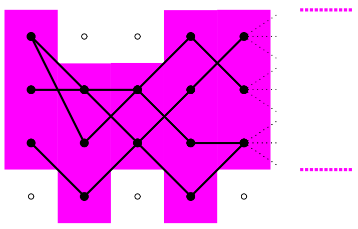


Fig. 4. Existential reachable state abstraction

Theorem 3. $\mathcal{G}^{\exists r} \stackrel{\text{def}}{=} (\alpha^{\exists r}_{M}, \wp(\mathrm{Trace}(\Sigma))_{\subseteq}, \wp(\Sigma)_{\subseteq}, \gamma^{\exists r}_{M})$ *is a GC.*

A graphical example of existential reachable state abstraction is depicted in Figure 4. Also in this case, this is not a GI. In fact, by considering the transition system $\mathcal{T}_1$ in Figure 3, we have that $\gamma^{\exists r}_{M}$ is not 1-1: $\gamma^{\exists r}_{M}(\varnothing) = \gamma^{\exists r}_{M}(\{a\}) = \neg M$.

By duality, the *universal* reachable state abstraction is defined as follows:

– $\alpha^{\forall r}_{M}(T) \stackrel{\text{def}}{=} \neg(\alpha^{\exists r}_{M}(\neg(T))) = \lambda i \in \mathbb{N}.\{s \in \Sigma \mid M_{\downarrow s} \subseteq T\}$;
– $\gamma^{\forall r}_{M}(S) \stackrel{\text{def}}{=} \neg(\gamma^{\exists r}_{M}(\neg S)) = \{\pi \in M \mid \exists i \in \mathbb{N}.\pi(i) \in S\}$.

Hence, $\mathcal{G}^{\forall r} = (\alpha^{\forall r}_{M}, \wp(\mathrm{Trace}(\Sigma))_{\supseteq}, \mathrm{Trace}(\wp(\Sigma))_{\sqsupseteq}, \gamma^{\forall r}_{M})$ is a GC as well.

4.5 Comparing Trace Abstractions

It turns out that traces of sets of states are more precise than both branching and reachable states, while branching and reachable states abstractions are incomparable.

Proposition 2. $\mathcal{G}^{\forall t} \leq \mathcal{G}^{\forall}$ *and* $\mathcal{G}^{\forall t} \leq \mathcal{G}^{\forall r}$*. Also,* $\mathcal{G}^{\forall}$ *and* $\mathcal{G}^{\forall r}$ *are incomparable.*

5 Completeness of Trace Abstractions

5.1 Branching Abstraction

The maximum common fragment $\mathrm{LTL_{det}}$ of LTL and ACTL has been characterized by Maidl [14]:

$$\begin{aligned}\mathrm{LTL_{det}} \ni \varphi ::= p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid (p \wedge \varphi_1) \vee (\neg p \wedge \varphi_2) \mid \\ \mathrm{X}\varphi \mid \mathrm{U}(p \wedge \varphi_1, \neg p \wedge \varphi_2) \mid \mathrm{W}(p \wedge \varphi_1, \neg p \wedge \varphi_2)\end{aligned}$$

Maidl [14] shows that $\mathrm{LTL_{det}} = \mathrm{LTL} \cap \mathrm{ACTL}$, namely for any $\varphi \in \mathrm{LTL}$, $\alpha^{\forall}_{M}(\llbracket\varphi\rrbracket) \in \mathrm{Sem}(\mathrm{ACTL})$ iff $\llbracket\varphi\rrbracket \in \mathrm{Sem}(\mathrm{LTL_{det}})$. This result is important in abstract model checking because formulae in $\mathrm{LTL} \cap \mathrm{ACTL}$ admit linear counterexamples.

It turns out that the branching abstraction is complete for all the logical/temporal operators of $\mathrm{LTL_{det}}$.

Theorem 4. *$\alpha^{\forall}_{M}$ is complete for the logical/linear operators in* $\boldsymbol{Op}_{\mathrm{LTL_{det}}}$.

Thus, as expected, by Theorem 1 we obtain that the branching abstraction is complete for $\mathrm{LTL_{det}}$ so that trace-based and state-based model checking of $\mathrm{LTL_{det}}$ are equivalent.

We also obtain a further consequence from this completeness result. Some attempts prior to Maidl's 2000 work [14] of characterizing LTL$\cap$ACTL considered the so-called *branchable* formulae of LTL: given $\varphi \in$ LTL, the formula $\varphi_{\mathrm{A}} \in$ ACTL is obtained from φ by preceding each linear temporal operator occurring in φ by the universal path quantifier A. A formula $\varphi \in$ LTL is branchable when $\alpha^{\forall}_{M}([\![\varphi]\!]) = [\![\varphi_{\mathrm{A}}]\!]$ [11,20]. We thus define $\mathrm{LTL_{br}} \overset{\mathrm{def}}{=} \{\varphi \in \mathrm{LTL} \mid \varphi \text{ is branchable}\}$. Since the abstract semantics $[\![\varphi]\!]^{\alpha^{\forall}_{M}}$ of a LTL formula φ exactly coincides with $[\![\varphi_{\mathrm{A}}]\!]$, by Theorem 4, we have that $\mathrm{LTL_{br}} = \{\varphi \in \mathrm{LTL} \mid \alpha^{\forall}_{M}([\![\varphi]\!]) = [\![\varphi]\!]^{\alpha^{\forall}_{M}}\}$. As a consequence of the above Theorem 4, of Theorem 1 and of Maidl's Theorem, we obtain the following alternative characterization relating $\mathrm{LTL_{det}}$ and $\mathrm{LTL_{br}}$.

Theorem 5. $\mathrm{Sem}(\mathrm{LTL_{det}}) = \mathrm{Sem}(\mathrm{LTL_{br}})$.

5.2 Trace of Sets Abstraction

The trace of sets abstraction $\alpha^{\forall t}_{M}$ is not complete for all the operators of LTL. This is indeed a consequence of a more general result in [16] stating that no refinement of the branching abstraction can be complete for all the operators of LTL. As an example, let us show that $\alpha^{\forall t}_{M}$ is not complete for disjunction. In fact, for the transition system $\mathcal{T}_2$ in Figure 3, by considering the set of traces (actually paths) $T_1 = \{ab^{\omega}, cd^{\omega}\}$ and $T_2 = \{ad^{\omega}, cb^{\omega}\}$, we have that:

$$\begin{aligned}
\alpha^{\forall t}_{M}(\neg T_1) \sqcup \alpha^{\forall t}_{M}(\neg T_2) &= \\
\langle\{b,d\},\{a,c\},\{a,c\},...\rangle \sqcup \langle\{b,d\},\{a,c\},\{a,c\},...\rangle &= \\
\langle\{b,d\},\{a,c\},\{a,c\},...\rangle &\sqsubset \\
\alpha^{\forall t}_{M}(\neg T_1 \cup \neg T_2) &= \\
\alpha^{\forall t}_{M}(\mathrm{Trace}(\Sigma)) &= \\
\langle\{a,b,c,d\},\{a,b,c,d\},\{a,b,c,d\},...\rangle &
\end{aligned}$$

On the other hand, it turns out that traces of sets are complete for conjunction, next and globally operators.

Proposition 3. *$\alpha^{\forall t}_{M}$ is complete for the following operators on* $\wp(\mathrm{Trace}(\Sigma))$*:* $\lambda S,T.S \cap T$, $\lambda T.\mathbf{X}(T)$, $\lambda T.\mathbf{G}(T)$.

Thus, by Theorem 1, we obtain that $\alpha^{\forall t}_{M}$ is complete for $\mathrm{L}(\{\wedge, \mathrm{X}, \mathrm{G}\})$. By duality, $\alpha^{\exists t}_{M}$ is complete for $\mathrm{L}(\{\vee, \mathrm{X}, \mathrm{F}\})$.

5.3 Reachable State Abstraction

As expected, also the reachable states abstraction is not complete for all the linear operators of LTL. For instance, let us show that $\alpha^{\forall r}_{M}$ is not complete for disjunction and next operators. As far as disjunction is concerned, let us consider the transition system $\mathcal{T}_3$ in Figure 3 and the set of traces (actually paths) $T_1 = \{ab^{\omega}\}$ and $T_2 = \{cb^{\omega}\}$.

$$\alpha^{\forall r}_{M}(\neg T_1) \cup \alpha^{\forall r}_{M}(\neg T_2) = \{c\} \cup \{a\} \neq \alpha^{\forall r}_{M}(\neg T_1 \cup \neg T_2) = \alpha^{\forall r}_{M}(\mathrm{Trace}(\Sigma)) = \{a,b,c\}.$$

Moreover, for the next operator $\mathbf{X}$ we have that:

$$\alpha_M^{\forall r}(\mathbf{X}(\neg\{ab^\omega\})) = \neg\alpha_M^{\exists r}(\mathbf{X}(\{ab^\omega\})) = \neg\alpha_M^{\exists r}(\{aab^\omega, bab^\omega, cab^\omega\}) = \neg\{a,b\} = \{c\}.$$

Conversely, we have that $\alpha_M^{\forall r}(\mathbf{X}(\gamma_M^{\forall r}(\alpha_M^{\forall r}(\neg\{ab^\omega\})))) = \neg\alpha_M^{\exists r}(\mathbf{X}(\gamma_M^{\exists r}(\alpha_M^{\exists r}(\{ab^\omega\}))))$ and $b^\omega \in \gamma_M^{\exists r}(\alpha_M^{\exists r}(\{ab^\omega\}))$ so that $cb^\omega \in \mathbf{X}(\gamma_M^{\exists r}(\alpha_M^{\exists r}(\{ab^\omega\})))$ and in turn $c \in \alpha_M^{\exists r}(\mathbf{X}(\gamma_M^{\exists r}(\alpha_M^{\exists r}(\{ab^\omega\}))))$, namely $c \notin \alpha_M^{\forall r}(\mathbf{X}(\gamma_M^{\forall r}(\alpha_M^{\forall r}(\neg\{ab^\omega\}))))$. We have thus shown incompleteness for $\mathbf{X}$:

$$\alpha_M^{\forall r}(\mathbf{X}(\neg\{ab^\omega\})) \neq \alpha_M^{\forall r}(\mathbf{X}(\gamma_M^{\forall r}(\alpha_M^{\forall r}(\neg\{ab^\omega\})))).$$

Here, it turns out that $\alpha_M^{\forall r}$ is complete for conjunction and globally operators.

Proposition 4. *$\alpha_M^{\forall r}$ is complete for the following operators on $\wp(\mathrm{Trace}(\Sigma))$: $\lambda S,T.S\cap T$ and $\lambda T.\mathbf{G}(T)$.*

Therefore, by Theorem 1, we obtain that $\alpha_M^{\forall r}$ is complete for $\mathrm{L}(\{\wedge, \mathrm{G}\})$. By duality, $\alpha_M^{\exists r}$ is complete for $\mathrm{L}(\{\vee, \mathrm{F}\})$.

6 Comparing Expressive Powers

The standard notion of expressive power recalled in Section 3.2 is based on the idea of comparing languages in a common domain of interpretation. In fact, given a domain of interpretation C of some semantic structure $\mathcal{S} = (C, AP, I)$ we can compare two languages L_1 and L_2 by comparing $\mathrm{Sem}_C(\mathrm{L}_1)$ and $\mathrm{Sem}_C(\mathrm{L}_2)$. Thus, we write $\mathrm{L}_1 \equiv_C \mathrm{L}_2$ ($\mathrm{L}_1 \leq_C/\geq_C \mathrm{L}_2$) when $\mathrm{Sem}_C(\mathrm{L}_1) = \mathrm{Sem}_C(\mathrm{L}_2)$ ($\mathrm{Sem}_C(\mathrm{L}_1) \supseteq/\subseteq \mathrm{Sem}_C(\mathrm{L}_2)$).

More in general, we can also compare languages in a common "abstract" domain of interpretation. For example, it is well known how to compare state languages in the abstract domain of partitions. Let L be a state language, namely to be evaluated on $\wp(\Sigma)$. Following Dams [6,7], the *logical equivalence* $\sim_{\mathrm{L}}$ on the state space Σ induced by L is defined as follows: $s_1 \sim_{\mathrm{L}} s_2$ iff $\forall\varphi \in \mathrm{L}.s_1 \in [\![\varphi]\!] \Leftrightarrow s_2 \in [\![\varphi]\!]$. The state partition associated to the equivalence $\sim_{\mathrm{L}}$ is here denoted by $P_{\mathrm{L}} \in \mathrm{Part}(\Sigma)$ and, following Dams [6,7], is called the *distinguishing power* of L. Then, two state languages L_1 and L_2 can be also compared according to their distinguishing power:

$$\mathrm{L}_1 \equiv_{\mathrm{Part}(\Sigma)} \mathrm{L}_2 \quad \text{iff} \quad P_{\mathrm{L}_1} = P_{\mathrm{L}_2}.$$

Of course, this is indeed an "abstract" way of comparing languages because

$$\mathrm{L}_1 \equiv_{\wp(\Sigma)} \mathrm{L}_2 \ \Rightarrow\ \mathrm{L}_1 \equiv_{\mathrm{Part}(\Sigma)} \mathrm{L}_2$$

while the reverse implication is obviously not true. The distinguishing power is more abstract than the expressive power because it is not able to discriminate presence/absence of negation: in fact, for any language L, if $\mathrm{L}^\neg$ denotes the language L plus negation then we have that $P_{\mathrm{L}} = P_{\mathrm{L}^\neg}$ [17]. As an example, let L_μ denotes mu-calculus. It is well known [1] that $P_{\mathrm{CTL}} = P_{\mathrm{L}_\mu}$, that is $\mathrm{CTL} \equiv_{\mathrm{Part}(\Sigma)} \mathrm{L}_\mu$, whereas $\mathrm{Sem}_{\wp(\Sigma)}(\mathrm{CTL}) \subsetneq \mathrm{Sem}_{\wp(\Sigma)}(\mathrm{L}_\mu)$, that is $\mathrm{L}_\mu \lneq_{\wp(\Sigma)} \mathrm{CTL}$. A number of further examples can be found in Dams' works [6,7].

The key point here is that the lattice $\mathrm{Part}(\Sigma)$ of state partitions is indeed an abstraction of the lattice of abstract domains $\mathrm{uco}(\wp(\Sigma))$, as shown in [17]. Starting from this observation, the idea of comparing languages at different levels of abstraction can be precisely formalized by abstract interpretation. Let us recall from [17] how $\mathrm{Part}(\Sigma)$ can be viewed as an abstraction of the lattice of abstract domains $\mathrm{uco}(\wp(\Sigma))$. We define the abstraction and concretization maps:

$$\mathrm{uco}(\wp(\Sigma)_{\subseteq})_{\sqsupseteq} \underset{\mathrm{par}}{\overset{\mathrm{pcl}}{\leftrightarrows}} \mathrm{Part}(\Sigma)_{\succeq}$$

where, for any $s \in \Sigma$ and $\mu \in \mathrm{uco}(\wp(\Sigma))$, $[s]_\mu \overset{\text{def}}{=} \{s' \in \Sigma \mid \mu(\{s'\}) = \mu(\{s\})\}$ and $\mathrm{par}(\mu) \overset{\text{def}}{=} \{[s]_\mu \mid s \in \Sigma\}$, while $\mathrm{pcl}(P) \overset{\text{def}}{=} \lambda X \in \wp(\Sigma). \cup \{B \in P \mid X \cap B \neq \varnothing\}$. Thus, two states belong to the same block of $\mathrm{par}(\mu)$ when they are abstracted by μ to the same set while $\mathrm{pcl}(P)(X)$ is the minimal covering of the set $X \subseteq \Sigma$ through blocks in P.Let us also remark that $\mathrm{pcl}(P)$ is a uco whose set of fixpoints is given by all the unions of blocks in P, i.e. $\mathrm{pcl}(P) = \{\cup_i B_i \mid \{B_i\} \subseteq P\}$. It turns out that $(\mathrm{par}, \mathrm{uco}(\wp(\Sigma))_{\sqsupseteq}, \mathrm{Part}(\Sigma)_{\succeq}, \mathrm{pcl})$ is a GI.

Let us observe that a state language L (to be evaluated on $\wp(\Sigma)$) which is closed under conjunction can be viewed as an abstract domain, in the sense that $\mathrm{Sem}_{\wp(\Sigma)}(\mathrm{L}) \in \mathrm{uco}(\wp(\Sigma)_{\subseteq})$ because $\mathrm{Sem}_{\wp(\Sigma)}(\mathrm{L})$ is meet-closed (cf. Section 3.1). Assume now that L is closed under conjunction. Then, the distinguishing power of L can be retrieved as an abstraction in $\mathrm{Part}(\Sigma)$ of the expressive power of L, that is $P_{\mathrm{L}} = \mathrm{par}(\mathrm{Sem}_{\wp(\Sigma)})$.

Obviously, this can be done in general for *any abstraction* $(\alpha, \mathrm{uco}(\wp(\Sigma))_{\sqsupseteq}, A, \gamma)$, namely we can define the α-expressive power of a state language L as the abstract value $\alpha(\mathrm{Sem}_{\wp(\Sigma)}) \in A$. Notice that $(\alpha, \mathrm{uco}(\wp(\Sigma))_{\sqsupseteq}, A, \gamma)$ is a generic *higher-order* abstract interpretation meaning that here we deal with an abstraction of the higher-order lattice of abstract domains of the concrete domain $\wp(\Sigma)$.

6.1 Generalizing the Linear vs. Branching Time Comparison

As recalled in Section 3.2, Emerson and Halpern [8] use the universal path quantifier $\forall$ for "abstracting" a linear time language L to a corresponding branching time language $\mathrm{B(L)} \overset{\text{def}}{=} \{\forall\varphi \mid \varphi \in \mathrm{L}\}$ so that the expressive power of L can be compared with that of any state language. As shown in Section 4.2, the path quantifier $\forall$ can be cast as the branching abstract interpretation $(\alpha_M^{\forall}, \wp(\mathrm{Trace}(\Sigma))_{\supseteq}, \wp(\Sigma)_{\supseteq}, \gamma_M^{\forall})$. Thus, the expressive power of B(L) (in $\wp(\Sigma)$) actually can be characterized as $\mathrm{Sem}_{\wp(\Sigma)}(\mathrm{B(L)}) = \{\alpha_M^{\forall}([\![\varphi]\!]) \mid \varphi \in \mathrm{L}\}$. Therefore, Emerson and Halpern [8] indeed define a mapping which abstracts the "trace" expressive power $\{[\![\varphi]\!] \mid \varphi \in \mathrm{L}\} \subseteq \mathrm{Trace}(\Sigma)$ of any linear time language L to a corresponding "branching time" expressive power $\{\alpha_M^{\forall}([\![\varphi]\!]) \mid \varphi \in \mathrm{L}\} \subseteq \Sigma$.

As noted above, when L is closed under conjunction it turns out that the expressive power $\mathrm{Sem}_{\wp(\mathrm{Trace}(\Sigma))}(\mathrm{L})$ of L is an abstract domain in $\mathrm{uco}(\wp(\mathrm{Trace}(\Sigma))_{\subseteq})$ because it is intersection-closed. Moreover, since $\alpha_M^{\forall} : \wp(\mathrm{Trace}(\Sigma))_{\supseteq} \to \wp(\Sigma)_{\supseteq}$ is an abstraction map and therefore preserves least upper bounds, it turns out that $\{\alpha_M^{\forall}([\![\varphi]\!]) \mid \varphi \in \mathrm{L}\}$ is intersection-closed as well, i.e., it is an abstract domain in $\mathrm{uco}(\wp(\Sigma)_{\subseteq})$. In our framework, this means that Emerson and Halpern define a mapping

$$\mathrm{EH}_{\forall} : \mathrm{uco}(\wp(\mathrm{Trace}(\Sigma))) \to \mathrm{uco}(\wp(\Sigma))$$

from trace abstract domains to branching state abstract domains. Thus, any trace abstraction $(\alpha, \wp(\mathrm{Trace}(\Sigma))_{\supseteq}, A, \gamma)$ allows us to generalize the $\mathrm{EH}_{\forall}$ transform from the specific branching abstraction $\alpha^{\forall}_{M}$ to the generic abstraction α: the generic transform $\mathcal{A}_{\alpha} : \mathrm{uco}(\wp(\mathrm{Trace}(\Sigma))) \to \mathrm{uco}(A)$ is therefore defined by $\mathcal{A}_{\alpha}(\mu) \stackrel{\mathrm{def}}{=} \{\alpha(T) \mid T \in \mu\}$. The interesting point is that $\mathcal{A}_{\alpha}$ gives rise to a higher-order abstract interpretation.

Theorem 6. *$\mathcal{A}_{\alpha}$ gives rise to a GI $(\mathcal{A}_{\alpha}, \mathrm{uco}(\wp(\mathrm{Trace}(\Sigma)))_{\sqsupseteq}, \mathrm{uco}(A)_{\sqsupseteq}, \mathcal{C}_{\alpha})$, where $\mathcal{C}_{\alpha}(\rho) \stackrel{\mathrm{def}}{=} \{T \subseteq \mathrm{Trace}(\Sigma) \mid \alpha(T) \in \rho\}$.*

In particular, the concretization functions $\mathrm{HE}_{\forall}\colon \mathrm{uco}(\wp(\Sigma)) \to \mathrm{uco}(\wp(\mathrm{Trace}(\Sigma)))$ which is right adjoint to Emerson and Halpern's transform $\mathrm{EH}_{\forall}$ is defined by

$$\mathrm{HE}_{\forall}(\rho) = \{T \subseteq \mathrm{Trace}(\Sigma) \mid \alpha^{\forall}_{M}(T) \in \rho\}.$$

Therefore, in order to compare a linear time language L and a branching time language $\mathcal{L}$ (both closed under conjunction) Emerson and Halpern compare the abstraction $\mathrm{EH}_{\forall}(\mathrm{Sem}_{\wp(\mathrm{Trace})}(\mathrm{L}))$ with $\mathrm{Sem}_{\wp(\Sigma)}(\mathcal{L})$. In our approach, given any abstraction $(\alpha, \wp(\mathrm{Trace}(\Sigma))_{\supseteq}, A, \gamma)$, like those in Section 4, we can compare L with any language $\mathcal{L}$ whose semantic evaluation is in A by comparing $\mathcal{A}_{\alpha}(\mathrm{Sem}_{\wp(\mathrm{Trace}(\Sigma))})$ with $\mathrm{Sem}_{A}(\mathcal{L})$.

Abstractions, namely Galois connections, can be composed. As an example, our abstract interpretation-based view allows to compose Emerson and Halpern's abstraction with the partitioning abstraction:

$$\begin{tikzcd} \mathrm{uco}(\wp(\mathrm{Trace}(\Sigma))_{\subseteq})_{\sqsupseteq} \arrow[r, two heads, shift right, "\mathrm{EH}_{\forall}"'] & \mathrm{uco}(\wp(\Sigma)_{\subseteq})_{\sqsupseteq} \arrow[l, shift right, "\mathrm{HE}_{\forall}"'] \arrow[r, two heads, shift right, "\mathrm{par}"'] & \mathrm{Part}(\Sigma)_{\succeq} \arrow[l, shift right, "\mathrm{pcl}"'] \end{tikzcd}$$

This is quite interesting because if L_1 and L_2 are comparable according to their expressive power they are also comparable according to their distinguishing power: this is an obvious consequence of the fact that abstraction maps are monotone. Thus, if L_1 and L_2 are incomparable in $\mathrm{Part}(\Sigma)$ they are also incomparable in $\wp(\Sigma)$. This can be helpful because comparisons in $\mathrm{Part}(\Sigma)$, i.e. based on distinguishing powers, could be easier than those in $\wp(\Sigma)$, i.e. based on expressive powers. In fact, one can compute the distinguishing power $P_{\mathcal{L}}$ of some state language $\mathcal{L}$ through a *partition refinement* algorithm. These can be efficient algorithms because they work by iteratively refining a current partition so that the number of iterations is always bounded by the height of the lattice $\mathrm{Part}(\Sigma)$, namely by $|\Sigma|$. Some well-known partition refinement algorithms are those by Paige and Tarjan [15] for CTL and by Groote and Vaandrager [10] for CTL-X. Moreover, there also exist partition refinement algorithms for generic state languages: see [6–Chapter 6] and [18]. Let us see an example.

Example 1. Let us consider the following two languages:

$$\mathrm{L} \ni \varphi ::= p \mid \mathrm{F}\varphi \mid \mathrm{G}\varphi \qquad\qquad \mathcal{L} \ni \psi ::= p \mid \mathrm{AF}\psi \mid \mathrm{AG}\psi$$

$\mathrm{L} \subseteq \mathrm{LTL}$ is linear time, $\mathcal{L} \subseteq \mathrm{ACTL}$ is branching time and we consider their standard interpretations. Notice that $\mathcal{L} = \mathrm{L}^{\forall}$, i.e., for any model M on a state space Σ, $\mathrm{Sem}_{\wp(\Sigma)}(\mathcal{L}) = \mathrm{Sem}_{\wp(\Sigma)}(\mathrm{L}^{\alpha^{\forall}_{M}})$. Our goal is comparing the expressive powers of $\forall\mathrm{L}$ and $\mathcal{L}$, i.e. $\mathrm{Sem}_{\wp(\Sigma)}(\alpha^{\forall}_{M}\mathrm{L})$ and $\mathrm{Sem}_{\wp(\Sigma)}(\mathcal{L})$. We show that they are incomparable by comparing their distinguishing powers $P_{\forall\mathrm{L}}$ and $P_{\mathcal{L}}$.

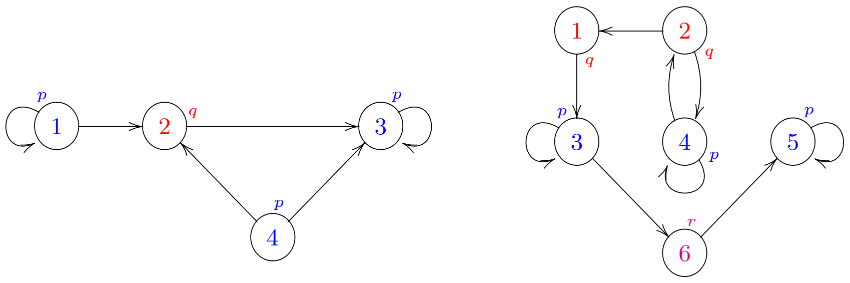


Fig. 5. Transition systems $\mathcal{T}_1$ (on the left) and $\mathcal{T}_2$ (on the right)

Let us consider the transition system $\mathcal{T}_1$ in Figure 5. Thus, the labelling for the atomic propositions p and q determines the initial partition $P = \{134, 2\}$. Let us characterize $P_{\mathfrak{L}}$. We have that $[\![\mathrm{AG}p]\!] = \{3\}$, so that P is refined to $P' = \{14, 2, 3\}$. Also, $[\![\mathrm{AFAG}p]\!] = \{2, 3, 4\}$, so that P' is refined to $P'' = \{1, 2, 3, 4\}$. Hence, $P_{\mathfrak{L}} = \{1, 2, 3, 4\}$. Let us now consider $\forall$L. Since $\mathrm{AG}p \in \forall\mathrm{L}$, also in this case P is first refined to $P' = \{14, 2, 3\}$. It turns out that this partition can be no more refined, because:

– $[\![\mathrm{AFG}p]\!] = [\![\mathrm{AGF}p]\!] = \{1, 2, 3, 4\}$; $[\![\mathrm{AFG}q]\!] = [\![\mathrm{AGF}q]\!] = \varnothing$;
– $\mathrm{FGF} = \mathrm{GF}$ and $\mathrm{GFG} = \mathrm{FG}$.

Thus, $P_{\mathfrak{L}} \prec P_{\forall\mathrm{L}}$.

Let us now consider the transition system $\mathcal{T}_2$ in Figure 5. Here, the labelling for the atomic propositions provides $P = \{12, 345, 6\}$ as initial partition. Let us characterize $P_{\mathfrak{L}}$. Since $[\![\mathrm{AG}p]\!] = \{5\}$, P is refined to $P' = \{12, 34, 5, 6\}$. This partition can be no more refined because:

$$[\![\mathrm{AF}\{12\}]\!] = \{1,2\},\ [\![\mathrm{AG}\{12\}]\!] = \varnothing; \qquad [\![\mathrm{AF}\{5\}]\!] = \{5,6\},\ [\![\mathrm{AG}\{5\}]\!] = \{5\};$$
$$[\![\mathrm{AF}\{34\}]\!] = \{1,2,3,4\},\ [\![\mathrm{AG}\{34\}]\!] = \varnothing; \qquad [\![\mathrm{AF}\{6\}]\!] = \{6\},\ [\![\mathrm{AG}\{6\}]\!] = \varnothing.$$

Thus, $P_{\mathfrak{L}} = \{12, 34, 5, 6\}$. On the other hand, let us characterize $P_{\forall L}$. In this case, it is enough to notice that $[\![\mathrm{AFG}p]\!] = \{1, 3, 5, 6\}$, so that P is refined to $P' = \{1, 2, 3, 4, 5, 6\}$. Hence, $P_{\forall\mathrm{L}} = \{1, 2, 3, 4, 5, 6\}$. In this case, we have that $P_{\forall\mathrm{L}} \prec P_{\mathfrak{L}}$.
Summing up, we showed that $\forall$L and $\mathfrak{L}$ are incomparable in $\mathrm{Part}(\Sigma)$, i.e., they have incomparable distinguishing powers. This implies that $\forall$L and $\mathfrak{L}$ have incomparable expressive powers. □

References

1. M.C. Browne, E.M. Clarke and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theor. Comp. Sci.*, 59:115-131, 1988.
2. E.M. Clarke and I.A. Draghicescu. Expressibility results for linear time and branching time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354:428-437, 1988.
3. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM POPL*, pp. 238-252, 1977.

4. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th ACM POPL*, pp. 269-282, 1979.
5. P. Cousot and R. Cousot. Temporal abstract interpretation. In *Proc. 27th ACM POPL*, pp. 12-25, 2000.
6. D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Ph.D. Thesis, Univ. Eindhoven, 1996.
7. D. Dams. Flat fragments of CTL and CTL*: separating the expressive and distinguishing powers. *Logic J. of the IGPL*, 7(1):55-78, 1999.
8. E.A. Emerson and J.Y. Halpern. “Sometimes” and “Not Never” revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151-178, 1986.
9. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361-416, 2000.
10. J.F. Groote and F. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proc. 17th ICALP*, LNCS 443:626-638, 1990.
11. O. Kupferman and M. Vardi. Relating linear and branching model checking. In *Proc. IFIP PROCOMET*, pp. 304-326, Chapman & Hall, 1998.
12. O. Kupferman and M. Vardi. Freedom, weakness and determinism: from linear-time to branching-time. In *Proc. 13th LICS*, pp. 81-92, IEEE Press, 1998.
13. L. Lamport. Sometimes is sometimes “not never” – on the temporal logic of programs. In *Proc. 7th ACM POPL*, pp. 174-185, 1980.
14. M. Maidl. The common fragment of CTL and LTL. In *Proc. 41st FOCS*, pp. 643-652, 2000.
15. R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973-989, 1987
16. F. Ranzato. On the completeness of model checking. In *Proc. 10th ESOP*, LNCS 2028:137-154, 2001
17. F. Ranzato and F. Tapparo. Strong preservation as completeness in abstract interpretation. In *Proc. 13th ESOP*, LNCS 2986:18-32, 2004.
18. F. Ranzato and F. Tapparo. An abstract interpretation-based refinement algorithm for strong preservation. In *Proc. 11th TACAS*, LNCS 3440:140-156, 2005.
19. M. Vardi. Sometimes and not never re-revisited: on branching vs. linear time. In *Proc. 9th CONCUR*, LNCS 1466:1-17, 1998.
20. M. Vardi. Branching vs. linear time: final showdown. In *Proc. 7th TACAS*, LNCS 2031:1-22, 2001.

The Parallel Implementation of the Astrée Static Analyzer

David Monniaux

Centre national de la recherche scientifique (CNRS), École normale supérieure,
Laboratoire d'Informatique, 45 rue d'Ulm, 75230 Paris cedex 5, France
David.Monniaux@ens.fr
http://www.di.ens.fr

Abstract. The ASTRÉE static analyzer is a specialized tool that can prove the absence of runtime errors, including arithmetic overflows, in large critical programs. Keeping analysis times reasonable for industrial use is one of the design objectives. In this paper, we discuss the parallel implementation of the analysis.

1 Introduction

The ASTRÉE static analyzer[1] is a tool that analyzes, fully automatically, single-threaded programs written in a subset of the C programming language, sufficient for many typical critical embedded programs. The tool particularly targets control/command applications using many floating-point variables and numerical filters, though it has been successfully applied to other categories of software. It computes a super-set of the possible run-time errors. ASTRÉE is designed for efficiency on large software: hundreds of thousands of lines of code are analyzed in a matter of hours, while producing very few false alarms. For example, some fly-by-wire avionics reactive control codes (70 000 and 380 000 lines respectively, the latter of a much more complex design) are analyzed in 1 h and 10 h 30' respectively on current single-CPU PCs, with *no false alarm* [1,2,9].

Other contributions [10,15,16,17,18,14] have described the *abstract domains* used in ASTRÉE; that is, the data structures and algorithms implementing the symbolic operations over abstract set of reachable states in the program to be analyzed. However, the operations in these abstract domains must be driven by an *iterator*, which follows the control flow of the program to be analyzed and calls the necessary operations. This paper describes some characteristics of the iterator. We first explain some peculiarities of our iteration algorithm as well as some implementation techniques regarding efficient shared data structures. These have an impact on the main contribution of the paper, which is the parallelization technique implemented in ASTRÉE.

Even though ASTRÉE presented good enough performances to be used in practical settings on large-scale industrial code on single-processor systems,

[1] http://www.astree.ens.fr

we designed a parallel implementation suitable both for shared-memory multiprocessor systems and for small clusters of PCs over local area networks. ASTRÉE being focused on synchronous, statically scheduled reactive programs, we used that peculiar form of the program to be analyzed in order to design a very simple, yet efficient, parallelization scheme. We however show that the control-flow properties that enable such parallel analysis are to be found in other kinds of programs, including major classes of programs such as event-driven user interfaces.

Section 2 describes the overall structure of the interpreter and the most significant choices about the iteration strategy. This defines the framework within which we implement our parallelization scheme. We also discuss implementation choices for some data structures, which have a large impact on the simplicity and efficiency of the parallel implementation.

Section 3 describes the parallelization of the abstract interpreter in a range of practical cases.

2 The ASTRÉE Abstract Interpreter

Our static analyzer is structured in several hierarchical layers:

– a denotational-based *abstract interpreter* abstractly executes the instructions in the programs by sending orders to the abstract domains;

– a *partitioning domain* [12] handles the partitioning of traces depending on various criteria; it also operates the partitioning with respect to the call stack;

– a *branching abstract domain* handles forward branching constructs such as forward `goto`, `break`, `continue`;

– a *structure abstract domain* resolves all accesses to complex data structures (arrays, pointers, records...) into may- or must-aliases over *abstract cells* [2–§6.1];

– various numerical domains express different kinds of constraints over those cells; each of these domains can query other domains for information, and send information to those domains (*reduction*).

2.1 A Denotational-Based Interpreter

Contrary to some presentations or examples of abstract interpretation-based static analysis [7] , we did not choose to obtain results through the direct resolution (or over-approximation) of a system of semantic equations, but rather to follow the denotational semantics of the program as in [6–Sect. 13].

Consider the following fragment of the C programming language:

$$\begin{array}{ll} l ::= x \mid t[e] \mid \dots & \text{l-values} \\ e ::= l \mid e \oplus e \mid \ominus e \mid \dots & \text{expressions } (\oplus \in \{+, \star, \dots\}; \ominus \in \{-, \dots\}) \\ s ::= \tau\ x; \mid l = e \mid \mathbf{if}(e)\{s; \dots; s\}\mathbf{else}\{s; \dots; s\} \mid \mathbf{while}(e)\{s; \dots; s\} & \end{array}$$

L is the set of control states, τ is any type, $\mathbb{L}$ (resp. $\mathbb{E}$) is the set of l-values (resp. expressions). The concrete semantics of a statement s is a function $[\![s]\!] : M \to \mathcal{P}(M) \times \mathcal{P}(E)$ where M (resp. E) is the set of memory states (resp. errors). Given an abstraction of sets of stores defined by an abstract domain $D^\sharp_M$

and a concretization function $\gamma_M : D^\sharp_M \to \mathcal{P}(M)$, we can derive an approximate abstract semantics $[\![P]\!]^\sharp : D^\sharp_M \to D^\sharp_M$ of program fragment P by following the methodology of abstract interpretation [8].

The soundness of $[\![P]\!]^\sharp$ can be stated as follows: if $[\![P]\!](\rho) = (m_0, e_0)$ and $\rho \in \gamma_M(d^\sharp)$, then $m_0 \subseteq \gamma_M([\![P]\!]^\sharp(d^\sharp))$ (resp. for the error list). The principle of the interpreter is to compute $[\![P]\!]^\sharp(d^\sharp)$ by induction on the syntax of P. $D^\sharp_M$ should provide abstract counterparts ($\mathbf{assign}, \mathbf{new\_var}, \mathbf{del\_var}, \mathbf{guard}$) to the concrete operations (assignment, variable creation and deletion, condition test). For instance, **assign** should be a function in $\mathbb{L} \times \mathbb{E} \times D^\sharp_M \to D^\sharp_M$, that inputs an l-value, an expression and an abstract value and returns a sound over-approximation of the set of stores resulting from the assignment: $\forall l \in \mathbb{L}, \forall e \in \mathbb{E}$, $\forall d^\sharp \in D^\sharp_M$, $\{\rho[[\![l]\!](\rho) \mapsto [\![e]\!](\rho)] \mid \rho \in \gamma_M(d^\sharp)\} \subseteq \gamma_M(\mathbf{assign}(l, e, d^\sharp))$. Soundness conditions for the other operations ($\mathbf{guard}, \mathbf{new\_var}, \mathbf{del\_var}$) are similar.

$$\begin{aligned}
[\![l = e;]\!]^\sharp(d^\sharp) &= \mathbf{assign}(l, e, d^\sharp) \quad \text{where } l \in \mathbb{L}, e \in \mathbb{E}\\
[\![\{\tau\, x; s_0\}]\!]^\sharp(d^\sharp) &= \mathbf{del\_var}(\tau\, x, [\![s_0]\!]^\sharp(\mathbf{new\_var}(\tau\, x, d^\sharp)))\\
[\![\mathbf{if}(e)\, s_0\, \mathbf{else}\, s_1;]\!]^\sharp(d^\sharp) &= [\![s_0]\!]^\sharp(\mathbf{guard}(e, \mathtt{t}, d^\sharp)) \sqcup [\![s_1]\!]^\sharp(\mathbf{guard}(e, \mathtt{f}, d^\sharp))\\
[\![\mathbf{while}(e)\, s_0]\!]^\sharp(d^\sharp) &= \mathbf{guard}(e, \mathtt{f}, \mathrm{lfp}^\sharp \phi^\sharp)\\
&\qquad \text{where: } \phi^\sharp : x^\sharp \in D^\sharp_M \mapsto d^\sharp \sqcup [\![s_0]\!]^\sharp(\mathbf{guard}(e, \mathtt{t}, x^\sharp))
\end{aligned}$$

The function $\mathrm{lfp}^\sharp$ computes a post-fixpoint of any abstract function (i.e., approximation of the concrete least-fixpoint). While the actual scheme implemented is somewhat complex, it is sufficient to say that $\mathrm{lfp}^\sharp f^\sharp$ outputs some $x^\sharp$ such that $f^\sharp(x^\sharp) \sqsubseteq^\sharp x^\sharp$ for some decidable ordering $\sqsubseteq^\sharp$ such that $\forall x^\sharp, y^\sharp\ x^\sharp \sqsubseteq^\sharp y^\sharp \implies \gamma(x^\sharp) \subseteq \gamma(y^\sharp)$. This abstract fixpoint is sought by the iterator in "iteration mode": possible warnings that could occur within the code are not displayed when encountered. Then, once $L^\sharp = \mathrm{lfp}^\sharp f^\sharp$ is computed — an invariant for the loop body —, the iterator analyzes the loop body again and displays possible warnings. As a supplemental safety measure, we check again that $f^\sharp(L^\sharp) \sqsubseteq^\sharp L^\sharp$.[2]

$\sqcup$ is an abstraction of the concrete union $\cup$: $\forall {d_1}^\sharp, {d_2}^\sharp, \gamma({d_1}^\sharp) \cup \gamma({d_2}^\sharp) \subseteq \gamma({d_1}^\sharp \sqcup\ {d_2}^\sharp)$.

An abstract domain handles the call stack; currently in ASTRÉE, it amounts to partitioning states by the full calling context [12]. ASTRÉE does not handle recursive functions;[3] this is not a problem with critical embedded code, since

[2] Let us note that the computationally costly part of the analysis is finding the loop invariant, rather than checking it. P. Cousot suggested the following improvement over our existing analysis: using different implementations for finding the invariant and checking it (at present, the same program does both). . For instance, the checking phase could be a possibly less efficient version, whose safety would be formally proved. However, since all abstract domains and most associated algorithms would have to be implemented in that "safe" analyzer, the amount of work involved would be considerable and we have not done it at this point. Also, as discussed in Sect. 3.2, both implementations would have to yield identical results, which means that the "safe" analysis would have to mimic the "unsafe" one in detail.

[3] More precisely, it can analyze recursive programs, but analysis may fail to terminate. If analysis terminates, then its results are sound.

programming guidelines for such systems generally prohibit the use of recursive functions. Functions are analyzed as if they were inlined at the point of call. Multiple targets for function pointers are analyzed separately, and the results merged with $\sqcup$; see §3 for an application to parallelization.

Other forms of branches are dealt with by an extension of the abstract semantics. Explicit gotos are rarely used in C, except as forward branches to error handlers or for exiting multiple loops; however, semantically similar branching structures are very usual and include **cases** structures, **break** statements and **return** statements. Indeed, a return statement **return** e carries out two operations: first, it evaluates e and stores the value as the function result; then, it terminates the current function, i.e. branches to the end of the function. In this paper, we only consider the case of forward-branching **goto**'s; the other constructs then are straightforward extensions.

We extend the syntax of statements with a goto statement **goto** l where l is a program point (we implicitly assume that there is a label before each statement in the program). The execution of a statement s may yield either a new memory state or a branching to a point after s. Therefore, we lift the definition of the semantics into a function $[\![s]\!] : (M \times (L \to \mathcal{P}(M))) \to (\mathcal{P}(M) \times (L \to \mathcal{P}(M)) \times E)$. The concrete states before and after each statement no longer consist solely of a set of memory states, but of a set of memory states for the "direct" control-flow as well as a set of memory states for each label l, representing all the memory states that have occurred in the past for which a forward branch to l was requested.

The concrete semantics of **goto** l is defined by: $[\![\textbf{goto}\ l;]\!](I_i, \phi_i) = (\bot, \phi_i[l \mapsto \phi_i(l) \cup I_i])$ and the concrete semantics of a statement s at label l is defined from the semantics without branches as: $[\![l : s;]\!](I_i, \phi_i) = (\{I_i\} \cup \phi_i(l), \phi_i)$. The definition of the abstract semantics can be extended in the same way. We straightforwardly lift the abstract semantics of a statement s into a function $[\![s]\!]^\sharp : D^\sharp_M \times (L \to D^\sharp_M) \to D^\sharp_M \times (L \to D^\sharp_M)$.

2.2 Rationale and Efficiency Issues

The choice of the denotational approach was made for two reasons:

- Iteration and widening techniques on general graph representations of programs are more complex. Essentially, these techniques partly have to reconstruct the natural control flow of the program so as to obtain an efficient propagation flow [3,11]. Since our programs are block-structured and do not contain backward `goto`'s, this flow information is already present in their syntax; there is no need to reconstruct it. [6–Sect. 13]
- It minimizes the amount of memory used for storing the abstract environments. While our storage methods maximize the sharing between abstract environments, our experiments showed that storing an abstract environment for each program point (or even each branching point) in main memory was too costly. Good forward/backward iteration techniques do not need to store environments at that many points, but this measurement still was an indication that there would be difficulties in implementing such schemes.

We measured the memory required for storing the local invariants at part or all of the program points, for three industrial control programs representative of those we are interested in; see the table below. We performed several measurements, depending on whether invariant data was saved at all statements, at the beginning and end of each block, and at the beginning and end of each function.

For each program and measurement, we provide two figures: from left to right, the peak memory observed during the analysis, then the size of the serialized invariant (serialization is performed for saving to files or for parallelization purposes, and preserves the sharing property of the internal representation).

Benchmarks (see below) show that keeping local invariants at the boundaries of every block in main memory is not practical on large programs; even restricting to the boundaries of functions results in a major overhead. A database system for storing invariants on secondary storage could be an option, but Brat and Venet have reported significant difficulties and complexity with that approach [4]. Furthermore, such an approach would complicate memory sharing, and perhaps force the use of solutions such as “hash-consing”, which we have avoided so far.

Memory requirements are expressed in megabytes; analyses were run in 64-bit mode on a dual Opteron 2.2 GHz machine with 8 Gb RAM.[4] On many occasions, we had to abort the computation due to large memory requirements causing the system to swap.

	Program 1		Program 2		Program 3	
# of lines of C code	67,553		232,859		412,858	
# of functions	650		1,900		2,900	
Save at all statements	3300	688	> 8000	swap	> 8000	swap
Save at beginning / end of blocks	2300	463	> 8000	swap	> 8000	swap
Save at beginning / end of functions	690	87	2480	264	4800	428
Save main loop invariant only	415	15	1544	53	2477	96
No save	410		1544		2440	

Benchmarks courtesy of X. Rival.

Memorizing invariants at the head of loops (the least set of invariants we can keep so as to be able to compute widening chains) thus entails much smaller memory requirements than a naïve graph-based implementation; the latter is intractable on reasonably-priced current computers on the class of large programs that we are interested in. It is possible that more complex memorization schemes may make graph-based algorithms tractable, but we did not investigate such schemes because we had an efficient and simple working system.

Regarding efficiency, it soon became apparent that a major factor was the efficiency of the $\sqcup$ operation. In a typical program, the number of tests will be roughly linear in the length of the code. In the control programs that ASTRÉE

[4] Memory requirements are smaller on 32-bit systems.

targets, the number of state variables (the values of which are kept across iterations) is also roughly linear in the length l of the code. This means that if the $\sqcup$ operation takes linear time in the number of variables — an apparently good complexity —, an iteration of the analyzer takes $\Theta(l^2)$ time, which is prohibitive. We therefore argue that *what matters is the complexity of* $\sqcup$ *with respect to the number of updated variables*, which should be almost linear: if only n_1 (resp. n_2) variables are touched in the `if` branch (resp. `else` branch), then the overall complexity should be at most roughly $O(n_1 + n_2)$.

We achieve such complexity with our implementation using balanced trees with strong memory sharing and "short-cuts" [1–§6.2]. Experimentation showed that memory sharing was good with the rough physical equality tests that we implement, without the need for much more costly techniques such as hash consing. Indeed, experiments show that considerable sharing is kept after the abstract execution of program parts that modify only parts of the global state (see the Δ-compression in §3.2). Though simple, this memory-saving technique is fragile; data sharing must be conserved by all modules in the program. This obligation had an impact on the design of the communications between parallel processes.

3 Parallelization

In iteration mode, we analyze tests in the following way: $[\![\mathbf{if}(e)\, s_0\, \mathbf{else}\, s_1;]\!]^\sharp(d^\sharp) = [\![s_0]\!]^\sharp(\mathbf{guard}(e, \mathtt{t}, d^\sharp)) \sqcup [\![s_1]\!]^\sharp(\mathbf{guard}(e, \mathtt{f}, d^\sharp))$. The analyses of s_0 and s_1 may be conducted in total separation, in different threads or processes, or even on different machines. Similarly, the semantics of an indirect function call may be approximated as: $[\![\mathtt{(*f)();}]\!]^\sharp(a^\sharp) = \bigsqcup_{g \in [\![\mathtt{f}]\!](a^\sharp)} [\![g]\!]^\sharp(a^\sharp)$: g ranges on all the possible code blocks to which `f` may point.

3.1 Dispatch Points

In usual programs, most tests split the control flow between short sequences of execution; the overhead of analyzing such short paths in separate processes would therefore be considerable with respect to the length of the analysis itself. However, there exist wide classes of programs where a few tests (at "dispatch points") divide the control flow between long executions. In particular, there exist several important kinds of software consisting in a large event loop: the system waits for an event, then a "dispatcher" runs an appropriate (often fairly complex) handler routine:

```
Initialization
while true do
  wait for a request $r$
  dispatch according to the type of $r$ in
    handler for requests of the first type
    handler for requests of the second type
    ...
done
```

This program structure is quite frequent in network services and traditional graphical user interfaces (though nowadays often wrapped inside a callback mechanism):

```
Initialization
**while true do**
  wait for an event $e$
  **dispatch** according to $e$ **in**
    event handler 1
    event handler 2
    ...
**done**
```

Many critical embedded programs are also of this form. We analyze reactive programs that, for the most part, implement in software a directed graph of numeric filters. Those numeric filters are in general the discrete-time counterparts of hardware, continuous-time components, with various sampling rates. The system is thus made of n components, each clocked with a period $p_i \cdot p$, where $1/p$ is a master clock rate (say, 1 kHz). It is statically scheduled as a succession of "sequencers" numbered from 0 to $N-1$ where N is the least common multiple of the p_i. A task of period $p_i.p$ is scheduled in all sequencers numbered $k.p_i + c_i$. c_i may often be arbitrarily chosen; judicious choices of c_i allow for static load balancing, especially with respect to worst-case execution time (all sequencers should complete within a time of p). Thus, the resulting program is of the form:

```
Initialization
**while true do**
  wait for clock tick ($1/p$ Hz)
  **dispatch** according to $i$ **in**
    sequencer 0
    sequencer 1
    ...
  $i := i + 1 \pmod{N}$
**done**
```

Our analysis is imprecise with respect to the succession of values of i; indeed, it soon approximates i by the full interval $[0, N-1]$. This is equivalent to saying that any of the sequencers is nondeterministically chosen. Yet, due to the nature of the studied system, this is not a hindrance to proving the absence of runtime errors: each of the n subcomponents should be individually sound, whatever the sampling rate, and the global stability of the system does not rely on the sampling rates of the subcomponents, within reasonable bounds.

Our system could actually handle parallelization at any place in the program where there exist two or more different control flows, by splitting the flows between several processors or machines; it is however undesirable to fork processes or launch remote analyses for simple blocks of code. Our current system decides the splitting point according to some simple ad-hoc criterion, but we could use some more universal criteria. For instance, the analyzer, during the first iteration(s), could measure the analysis times of all branches in `if`, `switch` or multi-aliases function calls; if a control flow choice takes place between several blocks for which the analysis takes a long time, the analysis could be split between several machines in the following iterations. To be efficient, however, such a system would have to do some relatively complex workload allocation between processors and machines; we will thus only implement it when really necessary.

3.2 Parallelization Implementation

Instead of analyzing all dispatch branches in sequence, we split the workload between p several processors (possibly in different machines). We replace the iterative computation of $f^\sharp(X^\sharp) = [\![P_1]\!]^\sharp(X^\sharp) \sqcup ([\![P_2]\!]^\sharp(X^\sharp \sqcup \ldots [\![P_n]\!]^\sharp(X^\sharp))\ldots)$ by a parallel computation $f^\sharp(X^\sharp) = \bigsqcup_{i=1}^{p}(\bigsqcup_{k\in\pi_i} [\![P_k]\!]^\sharp(X^\sharp))$ where the π_k are a partition of $\{1,\ldots,n\}$. Let us note τ_j the time needed to compute $[\![P_j]\!]^\sharp$. $l_k = \sum_{j\in\pi_k} \tau_j$ is the time spent by processor i.

For maximal efficiency, we would prefer that the l_i should be close to each other, so as to minimize the synchronization waits. Unfortunately, the problem of optimally partitioning into the π_k is NP-hard even in the case where $p = 2$ [13]. If the τ_i are too diverse, randomly shuffling the list may yield improved performance. In practice, the real-time programs that we analyze are scheduled so that all the P_i have about the same worst-case execution time, so as to ensure maximal efficiency of the embedded processor; consequently, the τ_i are reasonably close together and random shuffling does not bring significant improvement; in fact, in can occasionally reduce performances.

For large programs of the class we are interested in, the analysis times (Fig. 1) for n processors is approximately $0.75/n + 0.25$ times the analysis time on one processor; thus, clusters of more than 3 or 4 processors are not much interesting:

	Prog 1	Prog 2	Prog 3
# lines	67,553	232,859	412,858
1 CPU	26'28"	5h 55'	11h 30'
2 CPU	16'38"	3h 43'	7h 09'
3 CPU	14'47"	2h 58'	5h 50'
4 CPU	13'35"	2h 38'	5h 06'
5 CPU	13'26"	2h 25'	4h 44'

Venet and Brat also have experimented with parallelization [19–§5], with similar conclusions; however, the class of programs to be analyzed and the expected precisions of their analysis are too different from ours to make direct comparisons meaningful.

Because it is difficult to determine the τ_i in advance, ASTRÉE features an optional randomized scheduling strategy, which reduces computation times on our examples by 5%, with computation times on 2 CPUs $\simeq$58% of those on 1.

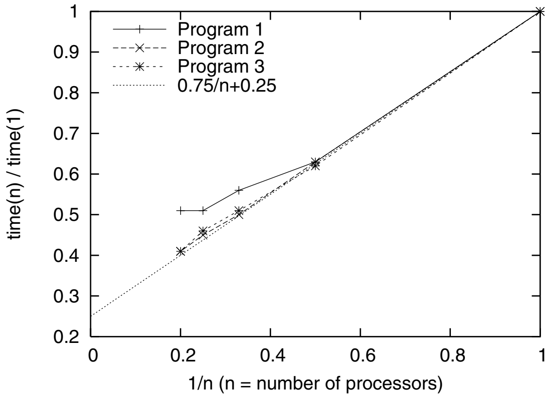


Fig. 1. Parallelization performances (dual-2.2 GHz Opteron machine + 2 GHz AMD64 machines)

We reduced transmission costs by sending only the differences between abstract values at the input and the output — when the remote computation is $[\![\mathrm{P}]\!]^\sharp(d^\sharp)$, only answer the difference between $d^\sharp$ and $[\![\mathrm{P}]\!]^\sharp(d^\sharp)$. This difference is obtained by physical comparison of data structures, excluding shared subtrees (Sect. 2.2). The advantage of that method is twofold:

– Experimentally, such "Δ-compression" results in transmissions of about 10% of the full size on our examples. This reduces transmission costs on networked implementations.
– Recall that we make analysis tractable by sharing data structures (Sect. 2.2). We however enforce this sharing by simple pointer comparisons (i.e. we do not construct another copy of a node if our procedure happens to have the original node at its disposal), which is fast and simple but does not guarantee optimal sharing. Any data coming from the network, even though logically equal to some data already present in memory, will be loaded at a different location; thus, one should avoid merging in redundant data. Sending only the difference back to the master analyzer thus dramatically reduces the amount of unshared data created by networked merge operations.

We request that the $\sqcup$ operator should be associative and commutative, so that $f^\sharp$ does not depend on the chosen partitioning. Such a dependency would be detrimental for two reasons:

– If the subprograms $P_1, \dots, P_n$ are enclosed within a loop, the nondeterminism of the abstract transfer function $f^\sharp$ complicates the analysis of the loop. As we said in Sect. 2.1, we use an "abstract fixpoint" operator $\mathrm{lfp}^\sharp$ that terminates when it finds $L^\sharp$ such that $f^\sharp(L^\sharp) \sqsubseteq L^\sharp$. Because this check is performed at least twice, it would be undesirable that the comparisons yield inconsistent results.

– For debugging and end-user purposes, it is undesirable that the results of the analysis could vary for the same analyzer and inputs because of runtime vagaries. [5]

In this case of a loop around the $P_1, \ldots, P_n$, we could have alternatively used asynchronous iterations [5]. To compute $\mathrm{lfp} f^\sharp$, one can use a central repository $X^\sharp$, initially containing $\bot$; then, any processor i computes $f_i^\sharp(X^\sharp) = \bigsqcup_{k \in \pi_i} [\![P_k]\!]^\sharp(X^\sharp)$ and replaces $X^\sharp$ with $X^\sharp \nabla f_i^\sharp(X^\sharp)$. If the scheduling is fair (no $[\![P_k]\!]$ is ignored indefinitely), such iterations converge to an approximation of the least fixpoint of $X \mapsto \cup_k [\![P_k]\!](X)$. However, we did not implement our analyzer this way. Apart from the added complexity, the nondeterminism of the results was undesirable.

4 Conclusion

We have investigated both theoretical and practical matters regarding the computation of fixpoints and iteration strategies for static analysis of single-threaded, block-structured programs, and proposed methods especially suited for the analysis of large synchronous programs: a denotational iteration scheme, maximal data sharing between abstract invariants, and parallelization schemes. In several occasions, we have identified possible extensions of our system.

Two major problems we have had to deal with were long computation times, and, more strikingly, large memory requirements, both owing to the very large size of the programs that we consider. Additionally, we had to keep a very high precision of analysis over complex numerical properties in order to be able to certify the absence of runtime errors in the industrial programs considered.

We think that several of these methods will apply to other classes of programs. Parallelization techniques, perhaps extended, should apply to wide classes

[5] For the same reasons, care should be exercised in networked implementations so that different platforms output the same analysis results on the same inputs. Subtle problems may occur in that respect; for instance, there may be differences between floating-point implementations. We use the native floating-point implementation of the analysis platform; even though all our host platforms are IEEE-compatible, the exact same analysis code may yield different results on various platforms, because implementations are allowed to provide more precision that requested by the norm. For example, the IA32$^{\mathrm{TM}}$(Intel Pentium$^{\mathrm{TM}}$) platform under Linux$^{\mathrm{TM}}$(and some other operating systems) computes by default internally with 80 bits of precision upon values that are specified to be 64-bit IEEE double precision values. Thus, the result of computations on that platform may depend on the register scheduling done by the compiler, and may also differ from results obtained on platforms doing all computations on 64-bit floating point numbers (PowerPC$^{\mathrm{TM}}$, and even IA32$^{\mathrm{TM}}$and AMD64$^{\mathrm{TM}}$with some code generation and system settings). Analysis results, in all cases, would be sound, but they would differ between implementations, which would be undesirable for the sake of reproducibility and debugging, and also for parallelization, as explained here. We thus force (if possible) the use of double (and sometimes single) precision IEEE floating-point numbers in all computations within the analyzer.

of event-driven programs; loop iteration techniques should apply to any single-threaded programs; data sharing and "union" optimizations should apply to any static analyzer. We also have identified various issues of a generic interest with respect to widenings and narrowings.

Acknowledgments. We wish to thank P. Cousot and X. Rival, as well as the rest of the ASTRÉE team.

References

1. B. Blanchet et al. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, number 2566 in LNCS, pages 85–108. Springer, 2002.
2. B. Blanchet et al. A static analyzer for large safety-critical software. In *PLDI*, 2003.
3. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. *LNCS*, 735:128, 1993.
4. G. Brat and A. Venet. Precise and scalable static program analysis of nasa flight software. In *IEEE Aerospace Conference*, 2005.
5. P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Technical Report 88, IMAG Lab., 1977.
6. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
7. P. Cousot and R. Cousot. Abstract intrepretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, Los Angeles, CA, January 1977.
8. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Logic Prog.*, 2-3(13):103–179, 1992.
9. P. Cousot et al. The ASTRÉE analyzer. In *ESOP*, volume 3444 of *LNCS*, 2005.
10. J. Feret. Static analysis of digital filters. In *ESOP*, volume 2986 of *LNCS*, 2004.
11. S. Horwitz, A. J. Demers, and T. Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.
12. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, volume 3444 of *LNCS*, 2005.
13. S. Mertens. The easiest hard problem: Number partitioning. In A.G. Percus, G. Istrate, and C. Moore, editors, *Computational Complexity and Statistical Physics*, page 8. Oxford University Press, 2004.
14. A. Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO*, volume 2053 of *LNCS*, 2001.
15. A. Miné. The octagon abstract domain. In *AST 2001*, IEEE, 2001.
16. A. Miné. A few graph-based relational numerical abstract domains. In *SAS*, volume 2477 of *LNCS*. Springer, 2002.
17. A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP*, volume 2986 of *LNCS*. Springer, 2004.
18. A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, 2004.
19. A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded C programs. In *PLDI*, 2004.

Using Datalog with Binary Decision Diagrams for Program Analysis

John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam

Computer Science Department, Stanford University,
Stanford, CA 94305, USA
{jwhaley, dzin, mcarbin, lam}@cs.stanford.edu

Abstract. Many problems in program analysis can be expressed naturally and concisely in a declarative language like Datalog. This makes it easy to specify new analyses or extend or compose existing analyses. However, previous implementations of declarative languages perform poorly compared with traditional implementations. This paper describes bddbddb, a BDD-Based Deductive DataBase, which implements the declarative language Datalog with stratified negation, totally-ordered finite domains and comparison operators. bddbddb uses binary decision diagrams (BDDs) to efficiently represent large relations. BDD operations take time proportional to the size of the data structure, not the number of tuples in a relation, which leads to fast execution times. bddbddb is an effective tool for implementing a large class of program analyses. We show that a context-insensitive points-to analysis implemented with bddbddb is about twice as fast as a carefully hand-tuned version. The use of BDDs also allows us to solve heretofore unsolved problems, like context-sensitive pointer analysis for large programs.

1 Introduction

Many program analyses can be expressed naturally and easily in logic programming languages, such as Prolog and Datalog [14,29,36]. Expressing a program analysis declaratively in a logic programming language has a number of advantages. First, analysis implementation is greatly simplified. Analyses expressed in a few lines of Datalog can take hundreds to thousands of lines of code in a traditional language. By automatically deriving the implementation from a Datalog specification, we introduce fewer errors. Second, because all analysis information is expressed in a uniform manner, it is easy to use analysis results or to combine analyses. Finally, optimizations of Datalog can be applied to all analyses expressed in the language.

However, implementations using logic programming systems are often slower than traditional implementations and can have difficulty scaling to large programs. Reps reported an experiment using Corel, a general-purpose logic programming system, to implement on-demand interprocedural reaching definitions analysis [29]. It was found that the logic programming approach was six times slower than a native C implementation. Dawson et al. used Prolog to perform groundness analysis on logic programs and strictness analysis on functional programs [14]. Using the XSB system, which has better efficiency than Corel [31], they were able to analyze a number of programs efficiently. However, the programs they analyzed were small — under 600 lines of code.

Other recent work by Liu and Stoller on efficient Datalog appears to have promise, but they do not present any performance results [21].

Our system for specifying program analyses using Datalog has successfully analyzed programs with tens of thousands of lines of source code, and has regularly performed faster than handcoded analyses. We will discuss our experience developing points-to analyses for C and for Java, which we compare with earlier handcoded versions. We will also discuss the results of using a security analysis and an external lock analysis which were specified using the same system.

1.1 Datalog Using BDDs

We have developed a system called bddbddb, which stands for BDD-Based Deductive DataBase. bddbddb is a solver for Datalog with stratified negation, totally-ordered finite domains and comparison operators. bddbddb represents relations using binary decision diagrams, or BDDs. BDDs are a novel data structure that were traditionally used for hardware verification and model checking, but have since spread to other areas. The original paper on BDDs is one of the most cited papers in computer science [8].

Recently, Berndl et al. showed that BDDs can be used to implement context-insensitive inclusion-based pointer analysis efficiently [5]. This work showed that a BDD-based implementation could be competitive in performance with traditional implementations. Zhu also investigated using BDDs for pointer analysis [40,41]. In 2004, Whaley and Lam showed that BDDs could actually be used to solve context-sensitive pointer analysis for large programs with an exponential number of calling contexts, a heretofore unsolved problem [38]. Thus, by using BDDs, we can solve new, harder program analysis problems for which there are no other known efficient algorithms.

Datalog is a logic programming language designed for relational databases. We translate each Datalog rule into a series of BDD operations, and then find the fixpoint solution by applying the operations for each rule until the program converges on a final set of relations. By using BDDs to represent relations, we can use BDD operations to operate on entire relations at once, instead of iterating over individual tuples.

Our goal with bddbddb was to hide most of the complexity of BDDs from the user. We have several years of experience in developing BDD-based program analyses and we have encoded our knowledge and experience in the design of the tool. Non-experts can develop their own analyses without having to deal with the complexities of fine-tuning a BDD implementation. They can also easily extend and build on top of the results of advanced program analyses that have been written for bddbddb.

Using bddbddb is not only easier than implementing an analysis by hand — it can also produce a more efficient implementation. bddbddb takes advantage of optimization opportunities that are too difficult or tedious to do by hand. We implemented Whaley and Lam's context-sensitive pointer analysis [38] using an earlier version of the bddbddb system and found that it performed significantly faster than a hand-coded, hand-tuned implementation based on BDDs. The hand-coded implementation, which was 100 times longer, also contained many more bugs.

We and others have also used bddbddb for a variety of other analyses and analysis queries, such as C pointer analysis, eliminating bounds check operations [1], finding security vulnerabilities in web applications [22], finding race conditions, escape analysis, lock analysis, serialization errors, and identifying memory leaks and lapsed listeners [23].

1.2 Contributions

This paper makes the following contributions:

1. Description of the bddbddb system. This paper describes in detail how bddbddb translates Datalog into efficient, optimized BDD operations and reports on the performance gains due to various optimizations. We expand upon material introduced in an earlier tutorial paper [18].
2. Demonstration of effective application of logic programming to problems in program analysis. Whereas previous work shows that there is a penalty in writing program analysis as database operations, we show that a BDD implementation of Datalog for program analysis can be very efficient. Interprocedural program analysis tends to create data that exhibits many commonalities. These commonalities result in an extremely efficient BDD representation. Datalog's evaluation semantics directly and efficiently map to BDD set operations.
3. Experimental results on a variety of program analyses over multiple input programs show that bddbddb is effective in generating BDD analyses from Datalog specifications. In particular, we compare bddbddb to some hand-coded, hand-optimized BDD program analyses and show that bddbddb is twice as fast in some cases, while also being far easier to write and debug.
4. Insights into using BDDs for program analysis. Before building this tool, we had amassed considerable experience in developing BDD-based program analyses. Much of that knowledge went into the design of the tool and our algorithms. This paper shares many of those insights, which is interesting to anyone who uses BDDs for program analysis.

1.3 Paper Organization

The rest of the paper is organized as follows. We first describe how a program analysis can be described as a Datalog program in Section 2. Section 3 deconstructs a Datalog program into operations in relational algebra, and shows how BDDs can be used to represent relations and implement relational operations. Section 4 describes the algorithm used by bddbddb to translate a Datalog program into an interpretable program of efficient BDD operations. Section 5 presents experimental results comparing bddbddb to hand-coded implementations of program analysis using BDDs. In Section 6 we discuss the related work. Our conclusions are in Section 7.

2 Expressing a Program Analysis in Datalog

Many program analyses, including type inference and points-to analyses, are often described formally in the compiler literature as inference rules, which naturally map to Datalog programs. A program analysis expressed in Datalog accepts an input program represented as a set of input relations and generates new output relations representing the results of the analysis.

2.1 Terminology

bddbddb is an implementation of Datalog with stratified negation, totally-ordered finite domains, and comparison operators. A Datalog program P consists of a set of domains

$\mathcal{D}$, a set of relations $\mathcal{R}$, and a set of rules $\mathcal{Q}$. The variables, types, code locations, function names, etc. in the input program are mapped to integer values in their respective domains. Statements in the program are broken down into basic program operations. Each type of basic operation is represented by a relation; operations in a program are represented as tuples in corresponding input relations. A program analysis can declare additional domains and relations. Datalog rules define how the new domains and relations are computed.

A domain $D \in \mathcal{D}$ has size $size(D) \in \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. We require that all domains have finite size. The elements of a domain D is the set of natural numbers $0 \ldots size(D) - 1$.

A relation $R \in \mathcal{R}$ is a set of n-ary tuples of *attributes*. The kth attribute of relation R is signified by $a_k(R)$, and the number of attributes of a relation R is signified by $arity(R)$. Relations must have one or more attributes, i.e. $\forall R \in \mathcal{R}, arity(R) \geq 1$. Each attribute $a \in \mathcal{A}$ has a domain $domain(a) \in \mathcal{D}$, which defines the set of possible values for that attribute. An expression $R(x_1, \ldots, x_n)$ is true iff the tuple $(x_1, \ldots, x_n)$ is in relation R. Likewise, $\neg R(x_1, \ldots, x_n)$ is true iff $(x_1, \ldots, x_n)$ is *not* in R.

Rules are of the form:

$$E_0 : -E_1, \ldots, E_k.$$

where the expression E_0 (the rule *head*) is of the form $R(x_1, \ldots, x_n)$, where $R \in \mathcal{R}$ and $n = arity(R)$. The expression list $E_1, \ldots, E_k$ (the rule *subgoals*) is a list of zero or more expressions, each with one of the following forms:

- $R(x_1, \ldots, x_n)$, where $R \in \mathcal{R}$ and $n = arity(R)$
- $\neg R(x_1, \ldots, x_n)$, where $R \in \mathcal{R}$ and $n = arity(R)$
- $x_1 = x_2$
- $x_1 \neq x_2$
- $x_1 < x_2$

The comparison expressions $x_1 = x_2$, $x_1 \neq x_2$, and $x_1 < x_2$ have their normal meanings over the natural numbers.

The domain of a variable x is determined by its usage in a rule. If x appears as the kth argument of an expression of the form $R(x_1, \ldots, x_n)$ then the domain of x, denoted by $domain(x)$, is $domain(a_k(R))$. All uses of a variable within a rule must agree upon the domain. Furthermore, in a comparison expression such as $x_1 = x_2$, $x_1 \neq x_2$ or $x_1 < x_2$, the domains of variables x_1 and x_2 must match.

A *safe* Datalog program guarantees that the set of inferred facts (relation tuples) will be finite. In bddbddb, because all domains are finite, programs are necessarily safe. If a variable in the head of a rule does not appear in any subgoals, that variable may take on any value in the corresponding attribute's domain; i.e. it will be bound to the universal set for that domain.

bddbddb allows negation in *stratifiable* programs [11]. Rules are grouped into strata, which are solved in sequence. Each strata has a *minimal solution*, where relations have the minimum number of tuples necessary to satisfy those rules. In a stratified program, every negated predicate evaluates the negation of a relation which was fully computed in a previous strata.

Datalog with *well-founded* negation is a superset of Datalog with stratifiable negation, and can be used to express fixpoint queries [15]. We have not yet found it necessary to extend bddbddb to support well-founded semantics, though it would not be difficult.

2.2 Example

Algorithm 1 is the Datalog program for a simple Java points-to analysis. It begins with a declaration of domains, their sizes, and optional mapping files containing meaningful names for the numerical values in each domain. V is the domain of local variables and method parameters, distinguished by identifier name and lexical scope. H is the domain of heap objects, named by their allocation site. F is the domain of field identifiers, distinguished by name and the type of object in which they are contained.

Relations are declared next, along with the names and domains of their attributes. Relation vP_0 is the set of initial points-to relations. vP_0 is declared as a set of tuples (v,h), where $v \in$ V and $h \in$ H. $vP_0(v,h)$ is true iff the program directly places a reference to heap object h in variable v in an operation such as `s = new String()`. Relation *store* represents store operations such as `x.f = y`, and *load* similarly represents load operations. $assign(x,y)$ is true iff the program contains the assignment `x=y`. Assuming that a program call graph is available *a priori*, intraprocedural assignments from method invocation arguments to formal method parameters and assignments from return statements to return value destinations can be modeled as simple assignments.

The analysis infers possible points-to relations between heap objects, and possible points-to relations from variables to heap objects. $vP(v,h)$ is true if variable v may point to heap object h at any point during program execution. Similarly, $hP(h_1,f,h_2)$ is true if heap object field $h_1.f$ may point to heap object h_2.

Rule 1 incorporates the initial points-to relations into vP. Rule 2 computes the transitive closure over inclusion edges. If variable v_2 can point to object h and v_1 includes v_2, then v_1 can also point to h. Rule 3 models the effect of store instructions on the heap. Given a statement $v_1.f = v_2$, if v_1 can point to h_1 and v_2 can point to h_2, then $h_1.f$ can point to h_2. Rule 4 resolves load instructions. Given a statement $v_2 = v_1.f$, if v_1 can point to h_1 and $h_1.f$ can point to h_2, then v_2 can point to h_2.

Algorithm 1 . Context-insensitive points-to analysis with a precomputed call graph, where parameter passing is modeled with assignment statements

Domains

V	262144	`variable.map`
H	65536	`heap.map`
F	16384	`field.map`

Relations

input	vP_0	$(variable : \mathrm{V}, heap : \mathrm{H})$
input	$store$	$(base : \mathrm{V}, field : \mathrm{F}, source : \mathrm{V})$
input	$load$	$(base : \mathrm{V}, field : \mathrm{F}, dest : \mathrm{V})$
input	$assign$	$(dest : \mathrm{V}, source : \mathrm{V})$
output	vP	$(variable : \mathrm{V}, heap : \mathrm{H})$
output	hP	$(base : \mathrm{H}, field : \mathrm{F}, target : \mathrm{H})$

Rules

$$vP(v,h) :- vP_0(v,h). \quad (1)$$
$$vP(v_1,h) :- assign(v_1,v_2), vP(v_2,h). \quad (2)$$
$$hP(h_1,f,h_2) :- store(v_1,f,v_2), vP(v_1,h_1), vP(v_2,h_2). \quad (3)$$
$$vP(v_2,h_2) :- load(v_1,f,v_2), vP(v_1,h_1), hP(h_1,f,h_2). \quad (4)$$

□

(a)	(b)
`String a = "fido";`	$vP_0(v_a, h_1)$
`String b;`	
`Dog d = new Dog();`	$vP_0(v_d, h_3)$
`b = a;`	$assign(v_b, v_a)$
`d.name = b;`	$store(v_d, name, v_b)$

Fig. 1. (a) Example program for Java pointer analysis. (b) Corresponding input relations.

To illustrate this analysis in action, we will use the simple Java program listed in Figure 1(a). Domain V contains values v_a, v_b and v_d representing variables `a`, `b`, and `d`. Domain H contains values h_1 and h_3, representing the objects allocated on lines 1 and 3. Domain F consists of the value $name$, which represents the `name` field of a `Dog` object.

The initial relations for this input program are given in Figure 1(b). Initial points-to relations in vP_0 are (v_a, h_1) and (v_d, h_3). The program has one assignment operation, represented as (v_b, v_a) in relation $assign$, and one store operation, represented as $(v_d, name, v_b)$ in relation $store$.

We begin by using Rule 1 to find that $vP(v_a, h_1)$ and $vP(v_d, h_3)$ are true. The results of the assignment on line 4 are found by using Rule 2, which tells us that $vP(v_b, h_1)$ is true since $assign(v_b, v_a)$ and $vP(v_a, h_1)$ are true. Finally, Rule 3 finds that $hP(h_3, name, h_1)$ is true, since $store(v_d, name, v_b)$, $vP(v_d, h_3)$, and $vP(v_b, h_1)$ are true.

3 From Datalog to BDD Operations

In this section, we explain our rationale for using BDD operations to solve Datalog programs. We first show how a Datalog program can be translated into relational algebra operations. We then show how we represent relations as boolean functions and relational algebra as operations on boolean functions. Finally, we show how boolean functions can be represented efficiently as binary decision diagrams (BDDs).

3.1 Relational Algebra

A Datalog query with finite domains and stratified negation can be solved by applying sequences of relational algebra operations corresponding to the Datalog rules iteratively, until a fixpoint solution is reached. We shall illustrate this translation simply by way of an example, since it is relatively well understood.

We use the following set of relational operations: join, union, project, rename, difference, and select. $R_1 \bowtie R_2$ denotes the *natural join* of relations R_1 and R_2, which returns a new relation where tuples in R_1 have been merged with tuples in R_2 in which corresponding attributes have equal values. $R_1 \cup R_2$ denotes the *union* of relations R_1 and R_2, which returns a new relation that contains the union of the sets of tuples in R_1 and R_2. $\pi_{a_1,\ldots,a_k}(R)$ denotes the *project* operation, which forms a new relation by removing attributes $a_1, \ldots, a_k$ from tuples in R. $\rho_{a \to a'}(R)$ denotes the *rename* operation, which returns a new relation with the attribute a of R renamed to a'. $R_1 - R_2$ denotes the *difference* of relations R_1 and R_2, which contains the tuples that are in R_1 but not in R_2. The *select* operation, denoted as $\sigma_{a=c}(R)$, restricts attribute a to match a constant value c. It is equivalent to performing a natural join with a unary relation consisting of a single tuple with attribute a holding value c.

To illustrate, an application of the rule

$$vP(v_1, h) : -assign(v_1, v_2), vP(v_2, h).$$

corresponds to this sequence of relational algebra operations:

$$\begin{array}{ll} t_1 & = \rho_{variable \rightarrow source}(vP); \\ t_2 & = assign \bowtie t_1; \\ t_3 & = \pi_{source}(t_2); \\ t_4 & = \rho_{dest \rightarrow variable}(t_3); \\ vP & = vP \cup t_4; \end{array}$$

Note that rename operations are inserted before join, union, or difference operations to ensure that corresponding attributes have the same name, while non-corresponding attributes have different names.

3.2 Boolean Functions

We encode relations as boolean functions over tuples of binary values. Elements in a domain are assigned consecutive numeric values, starting from 0. A value in a domain with m elements can be represented in $\lceil log_2(m) \rceil$ bits. Suppose each of the attributes of an n-ary relation R is associated with numeric domains $D_1, D_2, \ldots, D_n$, respectively. We can represent R as a boolean function $f : D_1 \times \ldots \times D_n \rightarrow \{0, 1\}$ such that $(d_1, \ldots, d_n) \in R$ iff $f(d_1, \ldots, d_n) = 1$, and $(d_1, \ldots, d_n) \notin R$ iff $f(d_1, \ldots, d_n) = 0$.

Let relation R be a set of tuples $\{(1, 1), (2, 0), (2, 1), (3, 0), (3, 1)\}$ over $D_1 \times D_2$, where $D_1 = \{0, 1, 2, 3\}$ and $D_2 = \{0, 1\}$. The binary encoding for R is function f, displayed in Figure 2(a), where the first attribute of R is represented by bits b_1 and b_2 and the second attribute by b_3.

For each relational algebra operation, there is a logical operation that produces the same effect when applied to the corresponding binary function representation. Suppose R_1 is represented by function $f_1 : D_1 \times D_2 \rightarrow \{0, 1\}$ and R_2 by function $f_2 : D_2 \times D_3 \rightarrow \{0, 1\}$. The relation $R_1 \bowtie R_2$ is represented by function $f_3 : D_1 \times D_2 \times D_3 \rightarrow \{0, 1\}$, where $f_3(d_1, d_2, d_3) = f_1(d_1, d_2) \wedge f_2(d_2, d_3)$. Similarly, the union operation maps to the binary $\vee$ operator, and $l - r \equiv l \wedge \neg r$. The project operation can be represented using existential quantification. For example, $\pi_{a_2}(R_1)$ is represented by $f : D_1 \rightarrow \{0, 1\}$ where $f(d_1) = \exists d_2 . f_1(d_1, d_2)$.

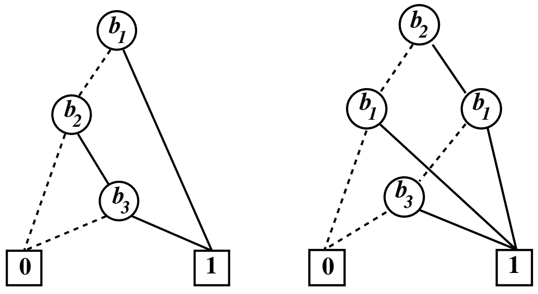
3.3 Binary Decision Diagrams

Large boolean functions can be represented efficiently using BDDs, which were originally invented for hardware verification to efficiently store a large number of states that share many commonalities [8].

A BDD is a directed acyclic graph (DAG) with a single root node and two terminal nodes which represent the constants one and zero. This graph represents a boolean function over a set of input decision variables. Each non-terminal node t in the DAG is labeled with an input decision variable and has exactly two outgoing edges: a high edge and a low edge. To evaluate the function for a given set of input values, one simply

D_1		D_2	R
b_1	b_2	b_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(a)



(b) (c)

Fig. 2. (a) Binary encoding of a relation. (b) and (c) are BDD encodings of the relation given by (a) with decision variable orders b_1, b_2, b_3 and b_2, b_1, b_3, respectively.

traces a path from the root node to one of the terminal nodes, following the high edge of a node if the corresponding input variable is true, and the low edge if it is false. The terminal node gives the value of the function for that input. Figure 2(b) shows a BDD representation for function f from Figure 2(a). Each non-terminal node is labeled with the corresponding decision variable, and a solid line indicates a high edge while a dashed line indicates a low edge.

We specifically use a variant of BDDs called *reduced ordered binary decision diagrams*, or ROBDDs [8]. In an *ordered* BDD, the sequence of variables evaluated along any path in the DAG is guaranteed to respect a given total *decision variable order*. The choice of the decision variable order can significantly affect the number of nodes required in a BDD. The BDD in Figure 2(b) uses variable order b_1, b_2, b_3, while the BDD in Figure 2(c) represents the same function, only with variable order b_2, b_1, b_3. Though the change in order only adds one extra node in this example, in the worst case an exponential number of nodes can be added. In addition, ROBDDs are *maximally reduced* meaning common BDD subgraphs are collapsed into a single graph, and the nodes are shared. Therefore, the size of the ROBDD depends on whether there are common boolean subexpressions in the encoded function, rather than on the number of entries in the set.

3.4 BDD Operations

The boolean function operations discussed in Section 3.2 are a standard feature of BDD libraries [20]. The $\wedge$ (and), $\vee$ (or), and $-$ (difference) boolean function operations can be applied to two BDDs, producing a BDD of the resulting function. The BDD existential quantification operation `exist` is used to produce a new BDD where nodes corresponding to projected attributes are removed. This operation combines the low and high successors of each removed node by applying an $\vee$ operation.

Rename operations are implemented using the BDD `replace` operation, which computes a new BDD where decision variables corresponding to the old attributes have been replaced with decision variables corresponding to the new attribute names. Replace operations can be eliminated if the renamed attributes are encoded using the same decision variables as the original attributes. A `replace` operation which does not change the relative order of decision variables is only linear with respect to the number of nodes in the BDD. If the order is changed, the cost of a `replace` can be exponential with respect to the number of decision variables. Care must be taken when encoding relation attributes to minimize the number of expensive rename operations.

Natural join operations are frequently followed by project operations to eliminate unnecessary attributes. The BDD relational product operation, or `relprod`, efficiently combines this sequence in a single operation. Similarly, the select and project operations can be combined into a single BDD operation, known as `restrict`.

BDD operations operate on entire relations at a time, rather than one tuple at a time. The cost of BDD operations depends on the size and shape of the BDD graphs, not the number of tuples in a relation. Thus, large relations can be computed quickly as long as their encoded BDD representations are compact. Also, due to caching in BDD packages, identical subproblems only have to be computed once. These points are key to the efficiency of BDD operations, and are the reason why we use this data structure to represent our relations.

4 Translating and Optimizing Datalog Programs

The bddbddb system applies a large number of optimizations to transform Datalog programs into efficient BDD operations:

1. Apply Datalog source level transforms and optimizations. (Section 4.1)
2. Remove unnecessary rules, stratify the rules, and determine the rule iteration order. (Section 4.2)
3. Translate the stratified query into an intermediate representation (IR) consisting of relational algebra operations. (Section 4.3)
4. Through analysis, optimize the IR and add BDD operations to replace equivalent sequences of relational algebra operations. (Section 4.4)
5. Choose BDD decision variables for encoding relation attributes. (Section 4.5)
6. Perform more dataflow optimizations after physical domains have been assigned. (Section 4.6)
7. Interpret the resulting program. (Section 4.7)

To illustrate this process, we use Algorithm 1 from Section 2 as a running example.

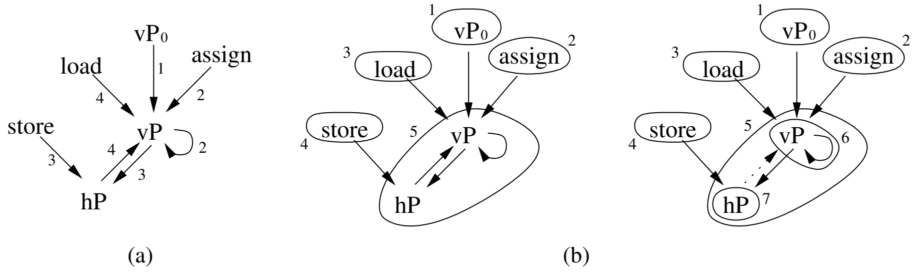


Fig. 3. (a) Predicate dependency graph for Algorithm 1. (b) Breaking the PDG into SCCs and finding cycles.

4.1 Datalog Source Transformations

Before compilation, we normalize the forms of the input rules as follows:

- Any variable that appears only once in a rule is changed into an underscore (_) to indicate an unused attribute.
- If a variable appears multiple times in a single subgoal, we give each additional use a distinct name, and then add extra equality subgoals to make the new variables equal to the original variable. For example, a subgoal $R(x, x, x)$ is transformed into the three subgoals $R(x, x', x''), x = x', x = x''$.
- Each comparison subgoal with an attribute of domain D is substituted with a subgoal for a corresponding precomputed relation defined over $D \times D$ which represents that comparison function.
- Subgoals in rules that define temporary relations are inlined into the rules that use those relations. Temporary relations are non-input, non-output relations which are in the head of only one rule, and appear as a subgoal in only one other rule.

4.2 Datalog Rule Optimization

Rule Removal. The solver removes rules and relations that do not indirectly contribute to the output relations. A *predicate dependency graph* (PDG) is built to record dependencies between rules and relations. Each node represents a relation, and there is an edge $g \rightarrow h$ marked with rule r if rule r has subgoal relation g and head relation h. (If the subgoal is negated, the edge is marked as a negative edge.) The PDG for our example is shown in Figure 3(a). Necessary rules and relations are found by performing a backward pass over the PDG, starting from the output relations.

Stratification. We then use the PDG to stratify the program. Stratification guarantees that the relation for every negated subgoal can be fully computed before applying rules containing the negation. Each stratum is a distinct subset of program rules that fully computes relations belonging to that stratum. Rules in a particular stratum may use the positive forms of relations computed in that stratum, as well as positive or negated forms of relations calculated in earlier strata and input relations from the relational database. There are no cyclic dependencies between strata. If the program cannot be stratified, we warn the user. In our experience designing Datalog programs for program analysis, we have yet to find a need for non-stratifiable queries.

As our example does not contain any negations, all of the rules and relations are placed within a single stratum.

Finding Cycles. Cycles in the PDG indicate that some rules and relations are recursively defined, requiring iterative application of rules within the cycles to reach a fixed-point solution. The PDG for each stratum is split into strongly connected components (SCCs). We can compute the result for a stratum by evaluating strongly connected components and non-cyclic relations in the topological order of the PDG.

A single strongly connected component can encompass multiple loops that share the same header node. We would like to distinguish between the different loops in a single SCC so we can iterate around them independently. However, the PDG is typically not reducible, and the classical algorithm for finding loops—Tarjan's interval finding algorithm—only works on reducible graphs [34]. Extensions have been made to deal with irreducible graphs, but they typically have the property that a node can only be the header for one loop [28]. We solve this by identifying one loop in the SCC, eliminating its back edge, and then recursively re-applying the SCC algorithm on the interior nodes to find more inner loops.

The steps of the algorithm on our example are shown in Figure **??**. We first break the PDG into five SCCs, labeled 1-5, as shown on the left. Then, we remove the edge for rule 4 from *hP* to *vP*, breaking the larger cycle so that it can topologically sort those nodes and find the smaller self-cycle on *vP* for rule 2, as shown on the right.

Determining Rule Application Order. The order in which the rules are applied can make a significant difference in the execution time. When there are multiple cycles in a single SCC, the number of rule applications that are necessary to reach a fixpoint solution can differ based on the relative order in which the two cycles are iterated. Which application order will yield the fewest number of rule applications depends not only on the rules but also on the nature of the relations.

Aspects of the BDD library can also make certain iteration orders more efficient than others, even if they have more rule applications. For example, the BDD library uses an operation cache to memoize the results of its recursive descents on BDD nodes, so it can avoid redundant computations when performing an operation. This cache can also provide benefits across different operations if the BDDs that are being operated upon share nodes. To take advantage of operation cache locality across operations, one should perform related operations in sequence. Another aspect influencing iteration order choice is the set-oriented nature of BDD operations. When performing an operation on tuples generated in a loop, it is ofter faster to apply the operation after completing all loop iterations, rather than applying it once per loop iteration.

In the absence of profile information from prior runs or from the user, bddbddb uses static analysis of the rules to decide upon a rule application order. Cycles that involve fewer rules are iterated before cycles that involve more rules, and rules that have fewer subgoals are iterated before rules that have more subgoals. The reasoning behind this is that smaller, shorter chains of rules and smaller rules are faster to iterate due to operation cache locality. This static metric works very well in the examples we have tried because small cycles are usually transitive closure computations, which are fast and expose more opportunities for set-based computation on the larger cycles.

4.3 Intermediate Representation

Once we have determined the iteration order, we translate the rules into an intermediate representation based on relational algebra operations as follows:

1. For each subgoal with an underscore, project away its unused attributes.
2. For each subgoal with a constant, use the select and project operators to restrict the relation to match the constant.
3. Join each subgoal relation with each of the other subgoal relations, projecting away attributes as they become unnecessary.
4. Rename the attributes in the result to match the head relation.
5. If the head relation contains a constant, use the select operator on the result to set the value of the constant.
6. Unify the result with the head relation.

4.4 IR Optimizations

In repeated applications of a given rule within a loop, it can be more efficient to make use of the differential between the current value of a subgoal relation and the previous value from the last time the rule was applied. This is known as *incrementalization* or the *semi-naïve* evaluation strategy. By computing the difference in subgoal relations as compared to the previous iteration, we can avoid extra work; if these inputs are the same as the last iteration, we can avoid applying the rule altogether. The tradeoff of incrementalization is that the old value of every subgoal in every incrementalized rule must be stored. We allow the user to control whether incrementalization is performed on a per rule basis. Performing incrementalization on the sequence of relational algebra operations derived from Rule (2) (Section 3.1) generates the following IR:

$$\begin{aligned}
vP'' &= vP - vP'; \\
vP' &= vP; \\
assign'' &= assign - assign'; \\
assign' &= assign; \\
t_1 &= \rho_{variable \to source}(vP''); \\
t_2 &= assign \bowtie t_1; \\
t_3 &= \rho_{variable \to source}(vP); \\
t_4 &= assign'' \bowtie t_3; \\
t_5 &= t_2 \cup t_4; \\
t_6 &= \pi_{source}(t_5); \\
t_7 &= \rho_{dest \to variable}(t_6); \\
vP &= vP \cup t_7;
\end{aligned}$$

Next, we apply a number of traditional compiler data flow optimizations on the IR:

- **Constant propagation.** We propagate empty set, universal set, and constants to reduce unions, joins, and difference operations.
- **Definition-use chains.** We calculate the chains of definitions and uses and use this to optimize the program by eliminating dead code (operations whose results have no uses), coalescing natural join and project pairs into `relprod` operations, and coalescing select and project pairs into `restrict` operations.

After this stage of optimizations, relational algebra operations are replaced by BDD operations, using combined `relprod` operations and `restrict` operations where possible. Rule (2) becomes:

$$\begin{aligned}
vP'' &= \mathtt{diff}(vP, vP'); \\
vP' &= \mathtt{copy}(vP); \\
t_1 &= \mathtt{replace}(vP'', variable \to source); \\
t_2 &= \mathtt{relprod}(t_1, assign, source); \\
t_3 &= \mathtt{replace}(t_2, dest \to variable); \\
vP &= \mathtt{or}(vP, t_3);
\end{aligned}$$

In the optimized IR, the join-project pair involving $assign$ and vP'' has been collapsed into a single `relprod`. Also, the operations for computing and using the difference of $assign$ have been removed because $assign$ is loop invariant.

4.5 BDD Decision Variable Assignment

As noted in Section 3.4, the use of BDD operations to implement relational operations places constraints on the choice of BDD decision variables used to encode relation attributes. When performing an operation on two BDDs, the decision variables for corresponding attributes must match. Likewise, unmatched attributes must be assigned to different decision variables. A BDD `replace` operation is used whenever different sets of decision variables must be substituted into a BDD as the result of a relational rename.

It is most important to minimize the cost of `replace` operations. This depends on the choice of decision variables used for encoding each attribute. The cost can be zero, linear, or exponential depending on whether the new decision variables are the same, have the same relative order, or have a different relative order. Additionally, we prefer to perform costly `replace` operations on smaller BDDs (in terms of BDD nodes) rather than on larger BDDs.

bddbddb uses a priority-based constraint system to assign attributes to BDD decision variables. This system is expressed in terms of both equivalence and non-equivalence constraints on relation attributes and sequences of decision variables. We use a specialized union-find data structure augmented with non-equivalence constraints to efficiently compute the constraint system. In BDD terminology, a sequence of binary decision variables used to represent an attribute is often referred to as a *physical domain*, which should not be confused with a Datalog domain as defined in Section 2.1.

We avoid introducing `replace` operations by constraining any renamed attributes to use the same physical domain as the original attribute. When an introduced constraint would cause the constraint system to have no solution, we assign the new attribute to a different physical domain and add a `replace` operation at that point to allow the constraint to be satisfied. By carefully choosing the order of priority in which constraints are added to the system, we ensure that `replace` operations are introduced where they will be most efficient.

For each attribute a in relation R, we create a non-equivalence constraint between a and other attributes in R. Then, we add constraints for all program operations, in order of importance. Operations in inner loops have higher importance than operations in outer loops, under the presumption that these operations will be performed more often. Within a given loop depth, `relprod` operations are considered first, in order of execution, since they are typically the most expensive operations. After `relprod` operations, we consider other operations. For a unary operation such as `copy`, we create equivalence constraints between corresponding attributes of the source and destination relations. For a binary operation, the interacting attributes for the input relations are constrained to be equal. After considering all operations, we add constraints for the attributes of the input and output relations. The physical domains used by these relations are specified by the user, since they must be loaded from or stored into the relational database.

An application of the physical domain assignment algorithm to our running example reveals that $\mathit{variable}$ from vP'' and source from assign can be assigned to the same physical domain for the `relprod`. Therefore, the `replace` that occurs immediately before can be removed:

$$\begin{aligned} vP'' &= \texttt{diff}(vP, vP'); \\ vP' &= \texttt{copy}(vP); \end{aligned}$$

$$\begin{aligned} t_1 &= \texttt{relprod}(vP'', assign, source); \\ t_2 &= \texttt{replace}(t_1, dest[V1] \rightarrow variable[V0]); \\ vP &= \texttt{or}(vP, t_2); \end{aligned}$$

4.6 Additional Optimizations

After domain assignment, we have the opportunity to apply another set of standard compiler optimizations:

- **Global value numbering.** Global value numbering factors the evaluation of common subexpressions among rules into non-redundant computations. Moreover, it optimizes loops by hoisting invariants.
- **Copy propagation.** Copy propagation eliminates unnecessary temporary IR relations that can be generated by our optimizations.
- **Liveness analysis.** We use a liveness analysis to clean up dead code. We reduce the memory footprint during IR interpretation by freeing relation allocations as soon as the lifetime of a relation has ended.

4.7 Interpretation

Finally, bddbddb interprets the optimized IR and performs the IR operations in sequence by calling the appropriate methods in the BDD library.

4.8 Decision Variable Ordering

While BDDs have proven effective in compacting the commonalities in large sets of data, the extent to which these commonalities can be exploited depends on the ordering of the decision variables. In our case, the difference between a good or bad ordering can mean the termination or non-termination (due to memory exhaustion) of an analysis. Moreover, the relative orderings are not readily apparent given only a static analysis, and the space of all orders is extremely large (with both precedence and interleaving conditions, the number of orders is given by the series for ordered Bell numbers).

We have developed an algorithm for finding an effective decision variable ordering [9]. The algorithm, based on active learning, is embedded in the execution of Datalog programs in the bddbddb system. When bddbddb encounters a rule application that takes longer than a parameterized amount of time, it initiates a learning episode to find a better decision variable ordering by measuring the time taken for alternative variable orderings. Because rule applications can be expensive, bddbddb maximizes the effectiveness of each trial by actively seeking out those decision variable orderings whose effects are least known.

5 Experimental Results

We measure the effectiveness of our bddbddb system and compare it to hand-optimized BDD programs. Prior to developing the bddbddb system, we had manually implemented and optimized three points-to analyses: a context-insensitive pointer analysis for Java described by Berndl [5], a context-sensitive pointer analysis based on the cloning of paths in the call graph [38], and a field-sensitive, context-insensitive pointer analysis for C [1]. We then wrote Datalog versions of these analyses which we ran using the bddbddb system.

The hand-coded Java analyses are the result of months of effort and are well-tuned and optimized. The variable ordering and physical domain assignment have been carefully hand-tuned to achieve the best results. Many of the rules in the hand-coded algorithms were incrementalized. This proved to be a very tedious and error-prone process, and we did not incrementalize the whole system as it would have been too unwieldy. Bugs were still popping up weeks after the incrementalization was completed. bddbddb, on the other hand, happily decomposed and incrementalized even the largest and most complex inference rules.

Because of the unsafe nature of C, the C pointer analysis is much more complicated, consisting of many more rules. For the hand-coded C pointer analysis, physical domain assignments, domain variable orderings and the order of inferences were only optimized to avoid significant execution slowdowns. Specification of low-level BDD operations was an error-prone, time-consuming process. A good deal of time was spent modifying physical domain assignments and solving errors due to the incorrect specification of physical domains in BDD operations. Once the Datalog version of the analysis was specified, development of the hand-coded version was discontinued, as it was no longer worth the effort. In the experiment reported here, we compare the hand-coded version and equivalent Datalog implementation from that time.

We also evaluate the performance of bddbddb on two additional analyses: an analysis to find external lock objects to aid in finding data races and atomicity bugs, and an analysis to find SQL injection vulnerabilities in Java web applications [23]. Both of these analyses build on top of the context-sensitive Java pointer analysis, and both are fairly sophisticated analyses. We do not have hand-coded implementations of these analyses as they would be too tedious to implement by hand.

5.1 Comparing Lines of Code

The first metric for comparison is in the number of lines of code in each algorithm:

Specifying the analysis as Datalog reduced the size of the analysis by 4.4 times in the case of the C analysis, to over 100 times in the case of the context-sensitive Java analysis. The disparity between the C and Java implementations is due to the fact that the C implementation combined many BDD operations on a single line, whereas the Java implementation put each BDD operation on a separate line of code.

Adding a new analysis with bddbddb takes only a few lines of code versus a rewrite of thousands of lines for a hand-coded implementation. The external lock analysis and the SQL injection analysis are examples of this. In another example, we easily modified

Analysis	**Hand-coded**	**Datalog**
context-insensitive Java	1975	30
context-sensitive Java	3451	33
context-insensitive C	1363	308
external lock analysis	n/a	42
SQL injection analysis	n/a	38

Fig. 4. LOC for hand-coded analyses versus lines of Datalog using bddbddb

the inference rules for the context-insensitive C points-to analysis to create a context-sensitive analysis by adding an additional context attribute to existing relations. While this was an extremely simple change to make to the bddbddb Datalog specification, such a modification would have required rewriting hundreds of lines of low-level BDD operations in the hand-coded analysis.

5.2 Comparing Analysis Times

For each analysis, we compared the solve time for an incrementalized hand-coded implementation against a bddbddb-based implementation with varying levels of optimization. Analyses were performed on an AMD Opteron 150 with 4GB RAM running RedHat Enterprise Linux 3 and Java JDK 5.0. The three bddbddb-based analyses and the hand-coded Java points-to analysis used the open-source JavaBDD library [37], which internally makes use of the BuDDy BDD library [20]. The hand-coded C points-to analysis makes direct use of the BuDDy library. The Java context-insensitive analysis used an initial node table size of 5M and an operation cache size of 750K. The Java context-sensitive analysis and C points-to analyses both used an initial node table size of 10M and an operation cache size of 1.5M.

Figures 5, 6 and 7 contain the run times of our Java context-insensitive analysis, Java context-sensitive analysis, and C pointer analysis, respectively. The first two columns give the benchmark name and description. The next column gives the solve time in seconds for the hand-coded solver. The remaining columns give the solve time when using bddbddb with various optimizations enabled. Each column adds a new optimization in addition to those used in columns to the left. Under **No Opts** we have all optimizations disabled. Under **Incr** we add incrementalization, as described in Section 4.3. Under **+DU** we add optimizations based on definition-use chains. Under **+Dom** we optimize physical domain assignments. Under **+All** we add the remaining optimizations described in Section 4. For the Java context-insensitive and C pointer analyses, the **+Order** column shows the result of bddbddb with all optimizations enabled using a variable order discovered by the learning algorithm referred to in Section 4.8. For our C programs, we used the order learned from `enscript`. For the Java programs we used the order learned from `joeq`. In the Java context-sensitive case, the learning algorithm was not able to find a better order, so we omitted this column. Entries marked with a ∞ signified that the test case did not complete due to running out of memory.

The time spent by bddbddb to translate Datalog to optimized BDD operations is negligible compared to the solve times, so the translation times have been omitted. In all cases, bddbddb spent no more than a few seconds to compile the Datalog into BDD operations.

The unoptimized context-insensitive Java analysis was 1.4 to 2 times slower than the hand-coded version. Incrementalization showed a very small improvement, but by adding def-use optimizations, we began to see a useful time reduction to 80% of the original. Optimizing BDD domain assignments reduces the runtime to about 42% of the original, and enabling all optimizations further reduces the runtime to about 38% of the original. Improved variable order brought the runtime between 24% and 36% of the unoptimized runtime. While incrementalization and def-use optimizations were sufficient to bring the bddbddb analysis close to the hand-coded analysis runtimes, the remaining optimizations and learned variable order combined to beat the hand-coded solver runtime by a factor of 2.

Name	Description	Hand-coded	bddbddb No Opts	Incr	+DU	+Dom	+All	+Order
joeq	virtual machine and compiler	7.3	10.0	9.4	7.9	4.8	4.5	3.6
jgraph	graph-theory library	15.0	25.6	24.1	20.0	11.0	10.4	7.6
jbidwatch	auction site tool	26.3	47.4	45.8	35.4	18.6	16.8	13.0
jedit	sourcecode editor	67.0	123.5	119.9	100.0	56.4	45.7	35.7
umldot	UML class diagrams from Java	16.6	29.0	27.4	20.2	11.6	10.9	8.4
megamek	networked battletech game	35.8	71.3	67.1	57.0	26.8	23.0	17.4

Fig. 5. Comparison of context-insensitive Java pointer analysis runtimes. Times are in seconds.

Name	Description	Hand-coded	bddbddb No Opts	Incr	+DU	+Dom	+All
joeq	virtual machine and compiler	85.3	323.3	317.8	274.7	124.7	69.7
jgraph	graph-theory library	118.0	428.1	431.1	362.2	116.3	94.9
jbidwatch	auction site tool	421.1	1590.2	1533.3	1324.3	470.6	361.3
jedit	sourcecode editor	147.0	377.2	363.4	293.7	136.4	109.3
umldot	UML class diagrams from Java	402.5	1548.3	1619.3	1362.3	456.5	332.8
megamek	networked battletech game	1219.2	∞	∞	4306.5	1762.9	858.3

Fig. 6. Comparison of context-sensitive Java pointer analysis runtimes. Times are in seconds.

Name	Description	Hand-coded	bddbddb No Opts	Incr	+DU	+Dom	+All	+Order
crafty	chess program	8.7	547.3	525.9	571.7	9.4	8.1	8.2
enscript	text to PS conversion	41.0	1175.4	1211.7	1128.4	122.3	112.6	31.5
hypermail	mbox to HTML conversion	149.4	6263.8	6113.0	5967.1	262.0	231.3	44.2
monkey	webserver	16.9	468.9	397.7	398.7	33.1	31.3	9.6

Fig. 7. Comparison of C pointer analysis runtimes. Times are in seconds.

Name	Description	bddbddb No Opts	Incr	+DU	+Dom	+All
joeq	virtual machine and compiler	75.0	60.4	59.3	17.4	15.1
jgraph	graph-theory library	64.9	51.0	51.1	13.0	12.5
jbidwatch	auction site tool	231.0	183.6	203.5	52.3	51.7
jedit	sourcecode editor	20.1	16.3	16.2	5.3	5.1
umldot	UML class diagrams from Java	199.3	162.2	161.3	45.0	39.2
megamek	networked battletech game	13.3	11.5	10.5	5.1	4.3

Fig. 8. External lock analysis runtimes. Times are in seconds.

Name	Description	bddbddb No Opts	Incr	+DU	+Dom	+All	+Order
personalblog	J2EE-based blogging application	∞	73.0	57.8	25.1	23.1	16.7
road2hibernate	hibernate testing application	∞	86.4	74.8	49.2	39.7	33.4
snipsnap	J2EE-based blogging application	∞	227.8	211.9	98.9	84.5	55.8
roller	J2EE-based blogging application	∞	521.0	479.0	253.7	208.4	185.4

Fig. 9. SQL injection query results. Times are in seconds. ∞ indicates that the analysis did not finish.

Results for the context-sensitive Java analysis were similar to the context-insensitive results. Unfortunately, our variable order learning algorithm was unable to learn a better variable order for this analysis, leaving the fully optimized bddbddb analysis about 20% faster than the hand-coded version.

In the case of the C analysis, the unoptimized bddbddb analysis was 23 to 60 times slower than the hand-coded version. This is likely due to the relative complexity of the Datalog in the C analysis case; optimizations were able to make significant improvements to the execution times. Analysis times with all optimizations enabled were roughly comparable to our hand-coded solver. As with the Java analyses, the largest gain was due to optimized physical domain assignment. When applying the learned variable order, bddbddb analysis runtimes were reduced even further, to fall between 30% and 95% of the hand-coded implementation.

5.3 External Lock and SQL Injection Analyses

We also used bddbddb to build external lock and SQL injection detection analyses on top of the Java points-to analysis results. The runtimes for the external lock analysis using different levels of optimization are displayed in Figure 8. Incrementalization reduces the analysis time to about 80% of the original time. Optimizing physical domain assignments further reduces the analysis time to about 23% of the original. Figure 9 displays the runtimes of the SQL injection analysis on four web-based applications. Without any incrementalization, the analysis fails to complete due to memory exhaustion. However, with further optimization we see performance gains similar to those of the external lock analysis.

6 Related Work

Related work falls into three general categories: optimizing Datalog executions, logic programming systems that use BDDs, and program analysis with BDDs. We go through each category in turn.

6.1 Optimizing Datalog

Liu and Stoller described a method for transforming Datalog rules into an efficient implementation based on indexed and linked data structures [21]. They proved their technique has “optimal” run time with respect to the fact that the combinations of facts that lead to all hypotheses of a rule being simultaneously true are considered exactly once. They did not present experimental results. Their formulation also greatly simplified the complexity analysis of Datalog programs. However, their technique does not apply when using BDDs, as the cost of BDD operations does not depend upon combinations of facts, but rather upon the number of nodes in the BDD representation and the nature of the relations.

There has been lots of research on optimizing Datalog evaluation strategies; for example, semi-naïve evaluation [10], bottom-up evaluation [10,24,35], top-down with tabling [12,33], etc. Ramakrishnan et al. investigated the role of rule ordering in computing fixpoints [26]. We use an evaluation strategy geared towards peculiarities of the BDD data structure — for example, to maximize cache locality, we iterate around inner loops first.

There has been work on transforming Datalog programs to reduce the amount of work necessary to compute a solution. Magic sets is a general algorithm for rewriting logical rules to cut down on the number of irrelevant facts generated [4]. This idea was extended to add better support for certain kinds of recursion [25]. Sagiv presented an algorithm for optimizing a Datalog program under uniform equivalence [30]. Zhou and Sato present several optimization techniques for fast computation of fixpoints by avoiding redundant evaluation of subgoals [39].

Halevy et al. describe the *query-tree*, a data structure that is useful in the optimization of Datalog programs [16]. The query-tree encodes all symbolic derivation trees that satisfy some property.

6.2 Logic Programming with BDDs

Iwaihara et al. described a technique for using BDDs for logic programming [17]. They presented two different ways of encoding relations: logarithmic encoding, which is the encoding we use in this paper, and linear encoding, which encodes elements or parts of elements as their own BDD variable. They evaluate the technique using a transitive closure computation. The Toupie system translates logic programming queries into an implementation based on decision diagrams [13].

Crocopat is a tool for relational computation that is used for structural analysis of software systems [7]. Like bddbddb, they use BDDs to represent relations.

6.3 Program Analysis with BDDs

Both Zhu and Berndl et al. used BDDs to implement context-insensitive inclusion-based points-to analysis [5,40]. Zhu extended his technique to support a summary-based context sensitivity [41], whereas Whaley and Lam developed a cloning-based context-sensitive pointer analysis algorithm that relies heavily on the data sharing inherent in BDDs [38]. Avots et al. extended Whaley and Lam's algorithm to support C programs with pointer arithmetic [1].

Jedd is a Java language extension that provides a relational algebra abstraction over BDDs [19]. Their treatment of domain assignment as a constraint problem is similar to ours; they use a SAT solver to find a legal domain assignment. They do not attempt to order the constraints based on importance.

Besson and Jensen describe a framework that uses Datalog to specify a variety of class analyses for object oriented programs [6]. Sittampalam, de Moor, and Larsen formulate program analyses using conditions on control flow paths [32]. These conditions contain free metavariables corresponding to program elements (such as variables and constants). They use BDDs to efficiently represent and search the large space of possible instantiations.

Bebop is a symbolic model checker used for checking program behavior [2]. It uses BDDs to represent sets of states. It has been used to validate critical safety properties of device drivers [3].

7 Conclusion

This paper described bddbddb, a deductive database engine that uses Datalog for specifying and querying program analyses. Datalog is a natural means of specifying many program analyses; many complicated analyses can be specified in only a few lines of

Datalog. Adding BDDs to this combination works well because BDDs can take advantage of the redundancies that occur in program analyses — especially context-sensitive analyses — and because BDD whole-set operations correspond closely to Datalog's evaluation style.

Our experience with the system is encouraging. Program analyses are so much easier to implement using bddbddb that we can no longer go back to the old technique of hand coding analyses. This is especially true because our experiments showed that bddbddb can often execute program analyses faster than a well-tuned handcoded implementation. Although there is still much work to be done in improving the algorithms and implementation of bddbddb, we have found the tool to be useful in our research.

The use of our system brings many benefits. It makes prototyping new analyses remarkably easy. Combining the results of multiple analyses becomes trivial. Concise specifications are easier to verify than larger traditional programs. The analysis runs faster because the inference engine automates the tedious process of optimizing and incrementalizing the analysis. New optimizations can be tested and implemented once in the inference engine, rather than repeatedly in each analysis. bddbddb bridges the gap between the specification of a program analysis and its implementation.

However, the greatest benefit of our system is that it makes powerful program analysis more widely accessible. The ease of a declarative language like SQL is considered to be one of the reasons for the success in databases [27]. We believe that the use of Datalog may play a important role in the future of interactive programming tools.

The bddbddb system is publicly available on Sourceforge licensed under the open-source LGPL license.

Acknowledgments

The authors would like to thank V. Benjamin Livshits and Christopher Unkel for their contribution of test cases and helpful feedback on their experiences with using bddbddb. This work was supported in part by the NSF under Grant No. 0326227 and an Intel Graduate Fellowship.

References

1. D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*. ACM Press, 2005.
2. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 113–130. Springer-Verlag, 2000.
3. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, pages 103–122. Springer-Verlag New York, Inc., 2001.
4. F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–15. ACM Press, 1986.
5. M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 103–114. ACM Press, 2003.

6. F. Besson and T. Jensen. Modular class analysis with datalog. In R. Cousot, editor, *Proceedings of the 10th Static Analysis Symposium (SAS 2003)*, pages 19–36. Springer LNCS vol. 2694, 2003.
7. D. Beyer, A. Noack, and C. Lewerentz. Simple and efficient relational querying of software structures. In *Proceedings of the 10th IEEE Working Conference on Reverse Engineering*, Nov. 2003.
8. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
9. M. Carbin, J. Whaley, and M. S. Lam. Finding effective variable orderings for BDD-based program analysis. To be submitted for publication, 2005.
10. S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases*. Springer-Verlag New York, Inc., 1990.
11. A. Chandra and D. Harel. Horn clauses and generalizations. *Journal of Logic Programming*, 2(1):1–15, 1985.
12. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.
13. M.-M. Corsini, K. Musumbu, A. Rauzy, and B. L. Charlier. Efficient bottom-up abstract interpretation of prolog by means of constraint solving over symbolic finite domains. In *PLILP '93: Proceedings of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 75–91. Springer-Verlag, 1993.
14. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systemsa case study. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 117–126. ACM Press, 1996.
15. A. V. Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, 1991.
16. A. Y. Halevy, I. S. Mumick, Y. Sagiv, and O. Shmueli. Static analysis in datalog extensions. *J. ACM*, 48(5):971–1012, 2001.
17. M. Iwaihara and Y. Inoue. Bottom-up evaluation of logic programs using binary decision diagrams. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 467–474. IEEE Computer Society, 1995.
18. M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, June 2005.
19. O. Lhoták and L. Hendren. Jedd: a BDD-based relational extension of Java. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 158–169. ACM Press, 2004.
20. J. Lind-Nielsen. BuDDy, a binary decision diagram package. http://buddy.sourceforge.net.
21. Y. A. Liu and S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 172–183. ACM Press, 2003.
22. V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *14th USENIX Security Symposium*. USENIX, Aug. 2005.
23. M. C. Martin, V. B. Livshits, and M. S. Lam. Finding application errors using PQL: a program query language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2005.
24. J. F. Naughton and R. Ramakrishnan. Bottom-up evaluation of logic programs. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 640–700, 1991.
25. J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 235–242. ACM Press, 1989.

26. R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 359–371. Morgan Kaufmann Publishers Inc., 1990.
27. R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *J. Logic Programming*, 23(2):125–149, 1993.
28. G. Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems*, 21(2):175–188, Mar. 1999.
29. T. W. Reps. *Demand Interprocedural Program Analysis Using Logic Databases*, pages 163–196. Kluwer, 1994.
30. Y. Sagiv. Optimizing datalog programs. In *PODS '87: Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 349–362. ACM Press, 1987.
31. K. Sagonas, T. Swift, and D. S. Warren. Xsb as an efficient deductive database engine. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 442–453. ACM Press, 1994.
32. G. Sittampalam, O. de Moor, and K. F. Larsen. Incremental execution of transformation specifications. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 26–38. ACM Press, 2004.
33. H. Tamaki and T. Sato. Old resolution with tabulation. In *Proceedings on Third International Conference on Logic Programming*, pages 84–98. Springer-Verlag New York, Inc., 1986.
34. R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9(3):355–365, Dec. 1974.
35. J. D. Ullman. Bottom-up beats top-down for datalog. In *PODS '89: Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 140–149. ACM Press, 1989.
36. J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, MD., volume II edition, 1989.
37. J. Whaley. JavaBDD library. http://javabdd.sourceforge.net.
38. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 131–144. ACM Press, 2004.
39. N.-F. Zhou and T. Sato. Efficient fixpoint computation in linear tabling. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 275–283. ACM Press, 2003.
40. J. Zhu. Symbolic pointer analysis. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 150–157. ACM Press, 2002.
41. J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 145–157. ACM Press, 2004.

Loop Invariants on Demand

K. Rustan M. Leino[1] and Francesco Logozzo[2]

[1] Microsoft Research, Redmond, WA, USA
leino@microsoft.com

[2] Laboratoire d'Informatique de l'École Normale Supérieure, F-75005 Paris, France
Francesco.Logozzo@polytechnique.edu

Abstract. This paper describes a sound technique that combines the precision of theorem proving with the loop-invariant inference of abstract interpretation. The loop-invariant computations are invoked on demand when the need for a stronger loop invariant arises, which allows a gradual increase in the level of precision used by the abstract interpreter. The technique generates loop invariants that are specific to a subset of a program's executions, achieving a dynamic and automatic form of value-based trace partitioning. Finally, the technique can be incorporated into a lemmas-on-demand theorem prover, where the loop-invariant inference happens after the generation of verification conditions.

1 Introduction

A central problem in reasoning about software is the infinite number of control paths and data values that a program's executions may give rise to. The general solution to this problem is to perform abstractions [10]. Abstractions include, for instance, predicates of interest on the data (as is done in predicate abstraction [21]) and summaries of the effects of certain control paths (like loop invariants [19,25]). A trend that has emerged in the last decade is to start with coarse-grained abstractions and to refine these when the need arises (as used, for example, in predicate refinement [21,3,24], lemmas-on-demand theorem provers [16,13,6,2,28], and abstract-interpretation based verifiers [32]).

In this paper, we describe a technique that refines loop invariants on demand. In particular, the search for stronger loop invariants is initiated as the need for stronger loop invariants arises during a theorem prover's attempt at proving the program. The technique can generate loop invariants that are specific to a subset of a program's executions, achieving a dynamic and automatic form of value-based trace partitioning. Finally, the technique can be incorporated into a lemmas-on-demand theorem prover, where the loop-invariant inference happens after the generation of verification conditions.

The basic idea is this: Given a program, we generate a *verification condition*, a logical formula whose validity implies the correctness of the program. We pass this formula to an automatic theorem prover that will either prove the correctness of the program or produce, in essence, a set of candidate program traces that lead to an error. Rather than just giving up and reporting these candidate traces as an error, we invoke an abstract interpreter on the loops along the traces, hoping to find stronger loop invariants that will allow the theorem prover to make more progress toward a proof. As this process continues, increasingly more precise analyses and abstract domains may be used with the abstract interpreter, which allows the scheme of optimistically trying cheaper analyses first. Each invocation of the abstract interpreter starts from the information available in the candidate traces. Consequently, the abstract interpreter can confine its analysis to these traces, thus computing loop invariants that hold along these traces but that may not hold for all of the program's executions. Once loop invariants are computed, they are communicated back to the theorem prover. This process terminates when the theorem prover is able to prove the program correct or when the abstract interpreter runs out of steam, in which case the candidate traces are reported.

As an example, consider the program in Figure 1. The strongest invariant for the loop is:

$$(N \leqslant 0 \ \wedge \ x = m = 0) \ \vee \ (0 \leqslant x \leqslant N \ \wedge \ 0 \leqslant m < N)$$

Using this loop invariant, one can prove the program to be correct, that is, one can prove that the assertion near the end of the program never fails. However, because this invariant contains a disjunction, common abstract domains like intervals [10], octagons [33], and polyhedra [12] would not be able to infer it. The strongest loop invariant that does not contain a disjunction is:

$$0 \leqslant x \ \wedge \ 0 \leqslant m$$

which, disappointingly, is not strong enough to prove the program correct. Disjunctive completion [11] or loop unrolling can be used to improve the precision of these domains, enough to prove the property. Nevertheless, in practice the cost of disjunctive completion is prohibitive and in the general case the number of loop unrollings necessary to prove the properties of interest is not known.

Trace partitioning is a well-known technique that provides a good value of the precision-to-cost ratio for handling disjunctive properties. Our technique performs trace partitioning dynamically (*i.e.*, during the analysis of the program), automatically, (*i.e.*, without requiring interaction from the user), and contextually (*i.e.*, according to the values of variables and the control flow of the program).

Applying our technique to the example program in Figure 1, the theorem prover would first produce a set of candidate traces that exit the loop with any values for x

```
$x := 0$ ;  $m := 0$ ;
while ($x < N$) {
   if (...) {          /* check if a new minimum as been found */
      $m := x$ ;
   }
   $x := x + 1$ ;
}
if ($0 < N$) {
   assert $0 \leqslant m < N$ ;
}
```

Fig. 1. A running example, showing a correct program whose correctness follows from a disjunctive loop invariant. The program is an abstraction of a program that iterates through an array (not shown) of length N (indexed from 0), recording in m the index of the array's mimimum element. The then branch of the if statement after the loop represents some operation that relies on the fact that m is a proper index into the array.

and m, then takes the then branch of the if statement (which implies $0 < N$) and then finds the assertion to be false. Not all such candidate trace are feasible, however. From the information about the candidate traces, and in particular from $0 < N$, the abstract interpreter (with the octagon or polyhedra domain, for example) infers the loop invariant:

$$0 \leqslant x \leqslant N \ \wedge \ 0 \leqslant m < N$$

using which the theorem prover can complete the proof.

In Section 2, we present a toy imperative language and define an abstract interpretation for it, parameterized by any abstract domain. We also prescribe for this language the generation of verification conditions. In Section 3, we present the basic interface of the theorem prover and define our technique as a "fact generator" for the theorem prover. Throughout, we apply what we say to the running example shown in Figure 1. We relate our technique to previous work in Section 4 and conclude the paper in Section 5.

2 An Imperative Language and Its Semantics

In this section, we define a while language, its abstract semantics, and its associated verification conditions. We also show an example program whose analysis benefits from the combination of an abstract interpreter and a theorem prover.

2.1 Grammar

We consider the source language whose grammar is given in Figure 2. The source language includes support for specifications via the **assert** E statement: if the expression E evaluates to false, then the program fails. The assignment statement $x := E$ sets

```
Stmt ::= assert Expr;                         (assertion)
       | x := Expr;                           (assignment)
       | havoc x;                             (set x to an arbitrary value)
       | Stmt Stmt                            (composition)
       | if (Expr) {Stmt} else {Stmt}         (conditional)
       | while^ℓ (Expr) {Stmt}                (loop)
```

Fig. 2. The grammar of the source language

```
x := 0;  m := 0;
while^ℓ (x < N) {
  havoc b;  if (b) {m := x; } else {assert true; }
  x := x + 1;
}
if (0 < N) {assert 0 ⩽ m < N; } else {assert true; }
```

Fig. 3. The program of Figure 1 encoded in the source language

the variable x to the value of the expression E. The havoc statement **havoc** x nondeterministically assigns a value to x. Sequential composition, conditionals, and loops are the usual ones. Note that we assume that loops are uniquely determined by labels ℓ taken from a set $\mathcal{W}$: Given a label ℓ, the function $\mathsf{LookupWhile}(\ell)$ returns the while loop associated with such a label.

For example, Figure 3 shows the program in Figure 1 written in the notation of our source language. Note the use of **havoc** b; followed by a use of b in a conditional, which encodes an arbitrary choice between the two branches of the conditional.

2.2 Abstract Semantics

The abstract semantics $\mathbb{c}[\![\cdot]\!]$ is defined by structural induction in Figure 4. It is parameterized by an abstract domain $\mathcal{D}$, which includes a join operator ($\sqcup$), a meet operator ($\sqcap$), a projection operator ($\mathsf{eliminate}$), and a primitive to handle the assignment (assign). The pointwise extension of the join is denoted $\dot{\sqcup}$.

The abstract semantics for expressions is given by a function $\mathbb{b}[\![\cdot]\!] \in \mathcal{L}(\mathit{Expr}) \to \mathcal{D} \to \mathcal{D}$. Intuitively, $\mathbb{b}[\![\mathrm{E}]\!](d)$ overapproximates the subset of the concrete states $\gamma(d)$ that make the expression E true. For lack of space, we omit here its definition and refer the interested reader to, *e.g.*, [9].

The input of the abstract semantics is an abstract state representing the initial conditions. The output is a pair consisting of an approximation of the output states and a map from (the label of) each loop sub-statement to an approximation of its loop invariant. The rules in Figure 4 are described as follows. The **assert** statement retains the part of the input state that satisfies the asserted expression. The effects of an assignment are handled by the abstract domain through the primitive assign. In the concrete, **havoc** x sets the variable x to any value, so in the abstract we handle it by simply projecting out the variable x from the abstract state. Stated differently, we set the value of x to $\top$. Sequential composition, conditional, and loops are defined as usual. In particular, the semantics of a loop is given by a least fixpoint on the abstract domain $\mathcal{D}$. Such a fixpoint can be computed iteratively, and if the abstract domain does not satisfy the

$$
\begin{aligned}
&\mathbb{c}[\![\cdot]\!] \in \mathcal{L}(\mathit{Stmt}) \to \mathcal{D} \to \mathcal{D} \times (\mathcal{W} \to \mathcal{D})\\
&\mathbb{c}[\![\mathbf{assert}\ \mathrm{E};]\!](d) \ = \ (\mathbb{b}[\![\mathrm{E}]\!](d),\ \emptyset)\\
&\mathbb{c}[\![x := \mathrm{E};]\!](d) \ = \ (d.\mathsf{assign}(x, \mathrm{E}),\ \emptyset)\\
&\mathbb{c}[\![\mathbf{havoc}\ x;]\!](d) \ = \ (d.\mathsf{eliminate}(x),\ \emptyset)\\
&\mathbb{c}[\![\mathrm{S}_0\ \mathrm{S}_1]\!](d) \ = \ \mathbf{let}\ (d_0, f_0) = \mathbb{c}[\![\mathrm{S}_0]\!](d)\ \mathbf{in}\\
&\qquad\qquad\qquad\quad \mathbf{let}\ (d_1, f_1) = \mathbb{c}[\![\mathrm{S}_1]\!](d_0)\ \mathbf{in}\\
&\qquad\qquad\qquad\quad (d_1,\ f_0 \cup f_1)\\
&\mathbb{c}[\![\mathbf{if}\ (\mathrm{E})\ \{\mathrm{S}_0\}\ \mathbf{else}\ \{\mathrm{S}_1\}]\!](d) \ = \ \mathbf{let}\ (d_0, f_0) = \mathbb{c}[\![\mathrm{S}_0]\!](\mathbb{b}[\![\mathrm{E}]\!](d))\ \mathbf{in}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ (d_1, f_1) = \mathbb{c}[\![\mathrm{S}_1]\!](\mathbb{b}[\![\neg\mathrm{E}]\!](d))\ \mathbf{in}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (d_0 \sqcup d_1,\ f_0 \cup f_1)\\
&\mathbb{c}[\![\mathbf{while}^{\ell}\ (\mathrm{E})\ \{\mathrm{S}\}]\!](d) \ =\\
&\qquad \mathbf{let}\ (d^*, f^*) = \mathsf{lfp}(\lambda X, Y \bullet\ (d, \emptyset)\ \dot{\sqcup}\ \mathbb{c}[\![\mathrm{S}]\!](\mathbb{b}[\![\mathrm{E}]\!](X))\ )\ \mathbf{in}\\
&\qquad (\mathbb{b}[\![\neg\mathrm{E}]\!](d^*),\ f^*[\ell \mapsto d^*])
\end{aligned}
$$

Fig. 4. The generic abstract semantics for the source language

$$\begin{array}{rll} \mathit{Cmd} ::= & \mathbf{assert}\ \mathit{Expr} & \text{(assert)} \\ \mid & \mathbf{assume}\ \mathit{Expr} & \text{(assume)} \\ \mid & \mathit{Cmd}\ ;\ \mathit{Cmd} & \text{(sequence)} \\ \mid & \mathit{Cmd}\ \square\ \mathit{Cmd} & \text{(non-deterministic choice)} \end{array}$$

Fig. 5. The intermediate language

ascending chain condition then the convergence (to a post-fixpoint) of the iterations is enforced through the use of a widening operator. The abstract state just after the loop is given by the loop invariant restrainted by the negation of the guard. Notice that the abstract semantics for the loop also records in the output map the loop invariant for ℓ.

2.3 Verification Conditions

To define the verification conditions for programs written in our source language, we first translate them into an intermediate language and then apply weakest preconditions (*cf.* [29]).

Intermediate Language. The commands of the intermediate language are given by the grammar in Figure 5. Our intermediate language is that of passive commands, *i.e.*, assignment-free and loop-free commands [18].

The **assert** and **assume** statements first evaluate the expression Expr. If it evaluates to true, then the execution continues. If the expression evaluates to false, the **assert** statement causes the program to fail (the program *goes wrong*) and the **assume** statement blocks the program (which implies that the program no longer has a chance of going wrong). Furthermore, we have a statement for sequential composition and non-deterministic choice.

The translation from a source language program S to an intermediate language program is given by the following function ($\mathbf{id}$ denotes the identity map):

$$\mathsf{translate}(\mathrm{S}) \;=\; \mathbf{let}\ (\mathrm{C}\,, m) = \mathsf{tr}(\mathrm{S}\,, \mathbf{id})\ \mathbf{in}\ \mathrm{C}$$

The goals of the translation are to get rid of (i) assignments and (ii) loops. To achieve (i), the translation uses a variant of static single assignment (SSA) [1] that introduces new variables (*inflections*) that stand for the values of program variables at different source-program locations, such that within any one execution path an inflection variable has only one value. To achieve (ii), the translation replaces an arbitrary number of iterations of a loop with something that describes the effect that these iterations have on the variables, namely the loop invariant. The definition of the function tr is in Figure 6. The function takes as input a program in the source language and a renaming function from program variables to their pre-state inflections, and it returns a program in the intermediate language and a renaming function from program variables to their post-state inflections. The rules in Figure 6 are described as follows.

The translation of an **assert** just renames the variables in the asserted expression to their current inflections. One of the goals of the passive form is to get rid of the assignments. As a consequence, given an assignment $x := \mathrm{E}$ in the source language, we generate a fresh variable for x (intuitively, the value of x after the assignment), we

```
$\mathsf{tr} \in \mathcal{L}(\mathit{Stmt}) \times (\mathtt{Vars} \rightarrow \mathtt{Vars}) \rightarrow \mathcal{L}(\mathit{Cmd}) \times (\mathtt{Vars} \rightarrow \mathtt{Vars})$
$\mathsf{tr}(\textbf{assert}\ \mathrm{E};,\ m)\ =\ (\textbf{assert}\ m(\mathrm{E}),\ m)$
$\mathsf{tr}(\mathtt{x} := \mathrm{E};,\ m)\ =\ (\textbf{assume}\ \mathtt{x}' = m(\mathrm{E}),\ m[\mathtt{x} \mapsto \mathtt{x}'])$ where $\mathtt{x}'$ is a fresh variable
$\mathsf{tr}(\textbf{havoc}\ \mathtt{x};,\ m)\ =\ (\textbf{assume}\ \mathit{true},\ m[\mathtt{x} \mapsto \mathtt{x}'])$ where $\mathtt{x}'$ is a fresh variable
$\mathsf{tr}(\mathrm{S}_0\ \mathrm{S}_1,\ m)\ =\ \textbf{let}\ (\mathrm{C}_0, n_0) = \mathsf{tr}(\mathrm{S}_0, m)\ \textbf{in}$
                       $\textbf{let}\ (\mathrm{C}_1, n_1) = \mathsf{tr}(\mathrm{S}_1, n_0)\ \textbf{in}$
                       $(\mathrm{C}_0\ ;\ \mathrm{C}_1,\ n_1)$
$\mathsf{tr}(\textbf{if}\ (\mathrm{E})\ \{\mathrm{S}_0\}\ \textbf{else}\ \{\mathrm{S}_1\},\ m)\ =$
    $\textbf{let}\ (\mathrm{C}_0, n_0) = \mathsf{tr}(\mathrm{S}_0, m)\ \textbf{in}$
    $\textbf{let}\ (\mathrm{C}_1, n_1) = \mathsf{tr}(\mathrm{S}_1, m)\ \textbf{in}$
    $\textbf{let}\ \mathtt{V} = \{\mathtt{x} \in \mathtt{Vars} \mid n_0(\mathtt{x}) \neq n_1(\mathtt{x})\}\ \textbf{in}$
    $\textbf{let}\ \mathtt{V}'\ \textbf{be}$ fresh variables for the variables in $\mathtt{V}$ $\textbf{in}$
    $\textbf{let}\ \mathrm{D}_0 = \textbf{assume}\ m(\mathrm{E})\ ;\ \mathrm{C}_0\ ;\ \textbf{assume}\ \mathtt{V}' = n_0(\mathtt{V})\ \textbf{in}$
    $\textbf{let}\ \mathrm{D}_1 = \textbf{assume}\ \neg m(\mathrm{E})\ ;\ \mathrm{C}_1\ ;\ \textbf{assume}\ \mathtt{V}' = n_1(\mathtt{V})\ \textbf{in}$
    $(\mathrm{D}_0 \square \mathrm{D}_1,\ m[\mathtt{V} \mapsto \mathtt{V}'])$
$\mathsf{tr}(\textbf{while}^{\ell}\ (\mathrm{E})\ \{\mathrm{S}\},\ m)\ =$
    $\textbf{let}\ \mathtt{V} = \mathsf{targets}(\mathrm{S})\ \textbf{in}$
    $\textbf{let}\ \mathtt{V}'\ \textbf{be}$ fresh variables for the variables in $\mathtt{V}$ $\textbf{in}$
    $\textbf{let}\ n = m[\mathtt{V} \mapsto \mathtt{V}']\ \textbf{in}$
    $\textbf{let}\ (\mathrm{C}, n_0) = \mathsf{tr}(\mathrm{S}, n)\ \textbf{in}$
    $\textbf{let}\ J_\ell\ \textbf{be}$ a fresh predicate symbol $\textbf{in}$
    $(\textbf{assume}\ J_\ell\langle \mathsf{range}(m),\ \mathsf{range}(n)\rangle\ ;$
     $(\textbf{assume}\ n(\mathrm{E})\ ;\ \mathrm{C}\ ;\ \textbf{assume}\ \mathit{false}\ \ \square\ \ \textbf{assume}\ \neg n(\mathrm{E})),\ \ n)$
```

Fig. 6. The function that translates from the source program to our intermediate language

apply the renaming function to E, and we output an **assume** statement that binds the new variable to the renamed expression. For instance, a statement that assigns to y in a state where the current inflection of y is $y0$ is translated as follows, where $y1$ is a fresh variable that denotes the inflection of y after the statement:

$$\mathsf{tr}(y := y + 4,\ [y \mapsto y0])\ =\ (\textbf{assume}\ y1 = y0 + 4,\ [y \mapsto y1])$$

The translation of **havoc** x just binds x to a fresh variable, without introducing any assumptions about the value of this fresh variable. The translation of sequential composition yields the composition of the translated statements and the post-renaming of the second statement. The translations of the conditional and the loop are trickier.

For the conditional, we translate the two branches to obtain two translated statements and two renaming functions. Then we consider the set of all the variables on which the renaming functions disagree (intuitively, they are the variables modified in one or both the branches of the conditional), and we assign them fresh names. These names will be the inflections of the variables after the conditional statement. Then, we generate the translation of the true (resp. false) branch of the conditional by assuming the guard (resp. the negation of the guard), followed by the translated command and the assumption of the fresh names for the modified variables. Finally, we use the non-deterministic choice operator to complete the translation of the whole conditional statement.

$$\begin{aligned}
&\mathsf{targets} \in \mathcal{L}(\mathit{Stmt}) \to \mathcal{P}(\mathtt{Vars})\\
&\mathsf{targets}(\textbf{assert}\ \mathrm{E};) \;=\; \emptyset\\
&\mathsf{targets}(\mathtt{x} := \mathrm{E};) \;=\; \mathsf{targets}(\textbf{havoc}\ \mathtt{x};) \;=\; \{\mathtt{x}\}\\
&\mathsf{targets}(\mathrm{S}_0\ \mathrm{S}_1) \;=\; \mathsf{targets}(\textbf{if}\ (\mathrm{E})\ \{\mathrm{S}_0\}\ \textbf{else}\ \{\mathrm{S}_1\}) \;=\; \mathsf{targets}(\mathrm{S}_0) \cup \mathsf{targets}(\mathrm{S}_1)\\
&\mathsf{targets}(\textbf{while}^{\ell}\ (\mathrm{E})\ \{\mathrm{S}\}) \;=\; \mathsf{targets}(\mathrm{S})
\end{aligned}$$

Fig. 7. The assignment targets, that is, the set of variables assigned in a source statement

```
**assume** $x_0 = 0$ ; **assume** $m_0 = 0$ ;
**assume** $J_\ell\langle(x_0, m_0, b, N),\ (x_1, m_1, b_0, N)\rangle$ ;
( **assume** $x_1 < N$ ;
  ( **assume** $b_1$ ; **assume** $m_2 = x_1$ ; **assume** $m_3 = m_2$
  $\square$ **assume** $\neg b_1$ ; **assert** $\mathit{true}$ ; **assume** $m_3 = m_1$
  ) ;
  **assume** $x_2 = x_1 + 1$ ; **assume** $\mathit{false}$
$\square$
  **assume** $\neg(x_1 < N)$
) ;
( **assume** $0 < N$ ; **assert** $0 \leqslant m_1 < N$
$\square$ **assume** $\neg(0 < N)$ ; **assert** $\mathit{true}$
)
```

Fig. 8. The intermediate-language program obtained as a translation of the source-language program in Figure 3. J_ℓ is a predicate symbol corresponding to the loop.

For the loop, we first identify the loop targets (defined in Figure 7), generate fresh names for them, and translate the loop body. Then, we generate a fresh predicate symbol indexed by the loop identifier (J_ℓ), which intuitively stands for the invariant of the loop $\mathsf{LookupWhile}(\ell)$. We output a sequence that is made up by the assumption of the loop invariant (intuition: we have performed an arbitrary number of loop iterations) and a non-deterministic choice between two cases: (i) the loop condition evaluates to true, we execute a further iteration of the body, and then we stop checking (**assume** *false*), or (ii) the loop condition evaluates to false and we terminate normally. Finally, please note that the arguments of the loop-invariant predicate J_ℓ include the names of program variables at the beginning of the loop ($\mathsf{range}(m)$) and the names of the variables after an arbitrary number of iterations of the loop ($\mathsf{range}(n)$).

Please note that we tacitely assume a total order on variables, so that, *e.g.*, the sets $\mathsf{range}(m)$ and $\mathsf{range}(n)$ can be isomorphically represented as lists of variables. We will use the list representation in our examples.

For example, applying $\mathsf{translate}$ to the program in Figure 3 results in the intermediate-language program shown in Figure 8.

Weakest Preconditions. The weakest preconditions of a program in the intermediate language are given in Figure 9, where $\varPhi$ denotes the set of first-order formulae. They characterize the set of pre-states from which every non-blocking execution of the command does not go wrong, and from which every terminating execution ends in a state

$$\begin{array}{l}
\mathsf{wp} \in \mathcal{L}(Cmd) \times \Phi \to \Phi \\
\mathsf{wp}(\mathbf{assert}\,\mathrm{E}\,,\ Q) \ = \ \mathrm{E} \ \wedge Q \\
\mathsf{wp}(\mathbf{assume}\,\mathrm{E}\,,\ Q) \ = \ \mathrm{E} \ \Rightarrow Q \\
\mathsf{wp}(\mathrm{C}_0\ ;\mathrm{C}_1,\ Q) \ = \ \mathsf{wp}(\mathrm{C}_0,\ \mathsf{wp}(\mathrm{C}_1, Q)) \\
\mathsf{wp}(\mathrm{C}_0 \,\square\, \mathrm{C}_1,\ Q) \ = \ \mathsf{wp}(\mathrm{C}_0, Q) \wedge \mathsf{wp}(\mathrm{C}_1, Q)
\end{array}$$

Fig. 9. Weakest preconditions of the intermediate language

$$\begin{array}{l}
x_0 = 0 \ \Rightarrow\ m_0 = 0 \ \Rightarrow \\
J_\ell\langle(x_0, m_0, b, N),\ (x_1, m_1, b_0, N)\rangle \ \Rightarrow \\
(x_1 < N \ \Rightarrow \\
\quad (b_1 \ \Rightarrow\ m_2 = x_1 \ \Rightarrow\ m_3 = m_2 \ \Rightarrow\ x_2 = x_1 + 1 \ \Rightarrow\ \mathit{false} \ \Rightarrow\ \ldots) \ \wedge \\
\quad (\neg b_1 \ \Rightarrow\ \mathit{true} \wedge (m_3 = m_1 \ \Rightarrow\ x_2 = x_1 + 1 \ \Rightarrow\ \mathit{false} \ \Rightarrow\ \ldots)) \\
) \ \wedge \\
(\neg(x_1 < N) \ \Rightarrow \\
\quad (0 < N \ \Rightarrow\ 0 \leqslant m_1 < N \ \wedge\ \mathit{true}) \ \wedge \\
\quad (\neg(0 < N) \ \Rightarrow\ \mathit{true} \ \wedge\ \mathit{true}) \\
)
\end{array}$$

Fig. 10. The weakest precondition of the program in Figure 8. We use $\Rightarrow$ as a right-associative operator with lower precedence than $\wedge$. The ellipsis in each of the two occurrences of the sub-formula "$\mathit{false} \ \Rightarrow\ \ldots$" stands for the conjunction shown in the second and third last lines.

satisfying Q [15,34]. As a consequence, the verification condition for a given program S, in the source language, is

$$\mathsf{wp}(\mathsf{translate}(\mathrm{S}), \mathit{true}) \qquad (1)$$

For example, the verification condition for the source program in Figure 3, obtained as the weakest precondition of the intermediate-language program in Figure 8, is the formula shown in Figure 10. (As can be seen in this formula, the verification condition contains a noticeable amount of redundancy, even for this small source program. We don't show it here, but the redundancy can be eliminated by using an important optimization in the computation of weakest preconditions, which is enabled by the fact that the weakest preconditions are computed from passive commands, see [18,27,4].)

2.4 The Benefit of Combining Analysis Techniques

It is unreasonable to think that every static analysis technique would encode all details of the operators (like integer addition, bitwise-or, and floating-point division) in a programming language. Operators without direct support can be encoded as uninterpreted functions. A theorem prover that supports quantifications offers an easy way to encode interesting properties of such functions. For example, the property that the bitwise-or of two non-negative integers is non-negative can be added to the analysis simply by including

$$(\forall\, x, y \bullet\ 0 \leqslant x \ \wedge\ 0 \leqslant y \ \Rightarrow\ 0 \leqslant \mathit{bitwiseOr}(x, y)\ ) \qquad (2)$$

in the antecedent of the verification condition. In an abstract interpreter, the addition of properties like this requires authoring or modifying an abstract domain, which takes

```
$x := 0$ ;
while$^\ell$ $(x < N)$ {
  if $(0 \leqslant y)$ {assert $0 \leqslant x \mid y$; } else {assert $true$; }
  $x := x + 1$;
}
```

Fig. 11. An artificial program whose analysis benefits from the combination of an abstract interpreter and a theorem prover

more effort. On the other hand, to use the theorem prover to prove a program correct, one needs to supply it with inductive conditions like invariants. An abstract interpreter computes (over-approximations of) such invariants. By combining an abstract interpreter and a theorem prover, one can reap the benefits of both the abstract interpreter's invariant computation and the theorem prover's high precision and easy extensibility.

For example, consider the program in Figure 11, where we use "$|$" to denote bitwise-or. Without a loop invariant, the theorem prover cannot prove that the first assertion holds. Without support for bitwise-or, an abstract interpreter cannot prove it either. But the combination can prove it: an abstract interpreter with support for intervals infers the loop invariant $0 \leqslant x$, and given this loop invariant and the axiom (2), a theorem prover can prove the program correct.

3 Loop-Invariant Fact Generator

To determine whether or not a program is correct with respect to its specification, we need to determine the validity of the verification condition (1), which we do using a theorem prover. A theorem prover can equivalently be thought of as a satisfier, since a formula is valid if and only if its negation is unsatisfiable. In this paper, we take the view of the theorem prover being a satisfier, so we ask it to try to satisfy the formula $\neg(1)$. If the theorem prover's exhaustive search determines that $\neg(1)$ is unsatisfiable, then (1) is valid and the program is correct. Otherwise, the prover returns a *monome*—a conjunction of possibly negated atomic formulas—that satisfies $\neg(1)$ and, as far as the prover can tell, is consistent. Intuitively, the monome represents a set of execution paths that lead to an error in the program being analyzed, together with any information gathered by the theorem prover about these execution paths.

A satisfying monome returned by the theorem prover may be an indication of an actual error in the program being analyzed. But the monome may also be the result of too weak loop invariants. (There's a third possibility: that the program's correctness depends on mathematical properties that are beyond the power or resource bounds of the prover. In this paper, we offer no improvement for this third possibility.) At the point where the prover is about to return a satisfying monome, we would like a chance to infer stronger loop invariants. To explain how this is done, let us give some more details of the theorem prover.

We assume the architecture of a lemmas-on-demand theorem prover [16,13]. It starts off viewing the given formula as a propositional formula, with each atomic sub-formula being represented as a propositional variable. The theorem prover asks a boolean-satisfiability (SAT) solver to produce monomes that satisfy the formula propositionally.

Each such monome is then scrutinized by the supported *theories*. These theories may include, for example, the theory of uninterpreted function symbols with equality and the theory of linear arithmetic. If the monome is found to be inconsistent with the theories, the theories generate a *lemma* that explains the inconsistency. The lemma is then, in effect, conjoined with the original formula and the search for a satisfying monome continues.

If the SAT solver finds a monome that is consistent with the theories, the theorem prover invokes a number of *fact generators* [28], each of which is allowed to return facts that may help refute the monome. For example, one such fact generator is the quantifier module, which Skolemizes existential quantifiers and heuristically instantiates universal quantifiers [17,28]. Unlike the lemmas generated by the theories, the facts may or may not be helpful in refuting the monome. Any facts generated are taken into consideration and the search for a satisfying monome is resumed. Only if the fact generators have no further facts to produce, or if some limit on the number of fact-generator invocations has been reached, does the theorem prover return the satisfying monome.

To generate loop invariants on demand, we therefore build a fact generator that infers loop invariants for the loops that play a role in the current monome. This fact generator can apply a more powerful technique with each invocation. For example, in a subsequent invocation, the fact generator may make use of more detailed abstract domains, it may perform more join operations before applying accelerating widen operations, or it may apply more narrowing operations. The fact generator can also make use of the contextual information of the current monome when inferring loop invariants—a loop invariant so inferred may not be a loop invariant in every execution of the program, but it may be good enough to refute the current monome.

The routine that generates new loop-invariant facts is shown in Figure 12. The GenerateFacts routine extracts from the monome each loop-invariant predicate $J_\ell\langle \mathtt{V}_0,\ \mathtt{V}_1\rangle$ of interest. The inference of a loop invariant for loop ℓ is done as follows.

First, GenerateFacts computes into d an initial state for the loop ℓ. This initial state is computed as a projection of the monome μ onto the loop pre-state inflections ($\mathtt{V}_0$). We let the abstract interpreter compute this projection, so we start by applying the abstraction function α, which maps the monome to an abstract element. For instance, using the polyhedra abstract domain, the α keeps just the parts of the monome that involve linear inequalities, all the other parts being abstracted away. The initial state, which is in terms of $\mathtt{V}_0$, is then conjoined with a set of equalities such that each program variable in $\mathtt{V}$ has the value of the corresponding variable in $\mathtt{V}_0$.

Then, GenerateFacts fetches the loop to be analyzed (LookupWhile(ℓ)) and runs the abstract interpreter with the initial state d. Since the abstract element d represents a relation on $\mathtt{V}_0$ and $\mathtt{V}$, the analysis will infer a relational invariant [30].

The abstract interpreter returns a loop invariant $f(\ell)$ with variables in $\mathtt{V}_0$ and $\mathtt{V}$. The routine GenerateFacts then renames each *program* variable (in $\mathtt{V}$) to its corresponding loop post-state inflection (in $\mathtt{V}_1$). Finally, the set of gathered facts is updated with the implication

$$\gamma(d_0)\ \wedge\ J_\ell\langle \mathtt{V}_0,\ \mathtt{V}_1\rangle\ \Rightarrow\ \gamma(\mathit{LoopInv})$$

Intuitively, it says that if the execution trace being examined satisfies d—the initial state of the loop used in the analysis just performed—then the loop-invariant predicate $J_\ell\langle \mathtt{V}_0,\ \mathtt{V}_1\rangle$ is no weaker than the inferred invariant $\mathit{LoopInv}$. In this formula, the abstract domain elements d and $\mathit{LoopInv}$ are first concretized using the concretization function γ, which produces a first-order formula in Φ.

Not utilized in Figure 12 are the invariants inferred for nested loops, which are also recorded in f. With a little more bookkeeping (namely, keeping track of the pre- and post-state inflections of the nested loop), one can also produce facts about these loops.

Continuing our example, when the negation of the verification condition in Figure 10 is passed to the theorem prover, the theorem prover responds with the following monome:

$$\begin{array}{l} x_0 = 0 \wedge m_0 = 0 \wedge \\ J_\ell\langle (x_0, m_0, b, N),\ (x_1, m_1, b_0, N)\rangle \wedge \\ \neg(x_1 < N) \wedge \\ 0 < N \wedge \neg(0 \leqslant m_1) \end{array}$$

(or, depending on how the SAT solver makes its case splits, the last literal in the monome may instead be $\neg(m_1 < N)$). When this monome is passed to GenerateFacts in Figure 12, the routine finds the $J_\ell\langle (x_0, m_0, b, N),\ (x_1, m_1, b_0, N)\rangle$ predicate, which tells it to analyze loop ℓ. We assume for this example that a numeric abstract domain like the polyhedra domain [12] is used. Since loop ℓ's pre-state inflections are (x_0, m_0, b, N), the abstract element d_0 is computed as

$$x_0 = 0 \ \wedge\ m_0 = 0 \ \wedge\ 0 < N$$

and d thus becomes

$$\begin{array}{l} x_0 = 0 \wedge m_0 = 0 \wedge 0 < N \wedge \\ x_0 = x \wedge m_0 = m \wedge b = b \wedge N = N \end{array}$$

The analysis of the loop produces in $f(\ell)$ the loop invariant

$$x_0 = 0 \leqslant x \leqslant N \wedge m_0 = 0 \leqslant m < N$$

```
GenerateFacts(Monome $\mu$) =
  let V be the program variables in
  var $\mathit{Facts} := \emptyset$ ;
  foreach $J_\ell\langle \mathtt{V}_0,\ \mathtt{V}_1\rangle \in \mu$ {
    var $d_0 := \alpha(\mu)$ ;
    foreach $\mathtt{x} \notin \mathtt{V}_0$ { $d_0 := d_0$.eliminate(x); }
    let $d = d_0 \sqcap \mathbb{b}[\![\mathtt{V}_0 = \mathtt{V}]\!](d_0)$ in
    let $(\_, f) = \mathbb{c}[\![\mathsf{LookupWhile}(\ell)]\!](d)$ in
    let $m = \mathtt{V} \rightarrow \mathtt{V}_1$ in
    let $\mathit{LoopInv} = m(f(\ell))$ in
      $\mathit{Facts} := \mathit{Facts} \cup \{\gamma(d_0) \wedge J_\ell\langle \mathtt{V}_0,\ \mathtt{V}_1\rangle \Rightarrow \gamma(\mathit{LoopInv})\}$ ;
  }
  return $\mathit{Facts}$
```

Fig. 12. The GenerateFacts routine, which invokes the abstract interpreter to infer loop invariants

Note that this loop invariant does not hold in general—it only holds in those executions where $0 < N$. Interestingly enough, notice that the condition $0 < N$ occurs *after* the loop in the program, but since it is relevant to the candidate error trace, it is part of the monome and thus becomes considered during the inference of loop invariants. Finally, the program variables are renamed to their post-state inflections (to match the second set of arguments passed to the J_ℓ predicate), which yields

$$x_0 = 0 \leqslant x_1 \leqslant N \wedge m_0 = 0 \leqslant m_1 < N$$

and so the generated fact is

$$\begin{array}{l} x_0 = 0 \wedge m_0 = 0 \wedge 0 < N \wedge J_\ell\langle(x_0, m_0, b, N),\ (x_1, m_1, b_0, N)\rangle \Rightarrow \\ x_0 = 0 \leqslant x_1 \leqslant N \wedge m_0 = 0 \leqslant m_1 < N \end{array}$$

With this fact as an additional constraint, the theorem prover is not able to satisfy the given formula. Hence, the program in Figure 3 has been automatically proved to be correct.

4 Related Work

Handjieva and Tzolovski [22] introduced a trace partitioning technique based on a program's control points. Their technique augments abstract states with an encoding of the history of the control flow. They consider finite sequences over $\{t_i, f_i\}$, where i is a control point and t_i (resp. f_i) denotes the fact that the true (resp. false) branch at the control point i is taken. Nevertheless, their approach abstracts away from the values of variables at control points, so that with such a technique it is impossible to prove correct the example in Figure 1.

The trace partitioning technique used by Jeannet *et al.* [26] allows performing the partition according the values of boolean variables or linear constraints appearing in the program text. Their technique is effective for the analysis of reactive systems, but in the general case it suffers from being too syntactic-based.

Bourdoncle considers a form of dynamic partitioning [7] for the analysis of recursive functions. In particular, he refines the solution of representing disjunctive properties by a set of abstract elements (roughly, the disjunctive completion of the underlying abstract domain), by limiting the number of disjuncts through the use of a suitable widening operator.

Mauborgne and Rival [32] present a systematic view of trace partitioning techniques, and in particular they focus on automatic partitioning for proving the absence of run-time errors in large critical embedded software. The main difference with our work is that in our case the partition is driven by the property to be verified, *i.e.*, the refuted verification condition.

Finally, the theoretical bases for trace partitioning are given by the reduced cardinal product (RDC) of abstract domains [11,20]. Roughly, the RDC of two abstract domains $\mathcal{D}_0$ and $\mathcal{D}_1$ produces a new domain made up of all the monotonic functions with domain $\mathcal{D}_0$ and co-domain $\mathcal{D}_1$. In trace partitioning, $\mathcal{D}_0$ contains the elements that allow the partitioning.

Yorsh *et al.* [36] and Zee *et al.* [37] use approaches different from ours for combining theorem proving and abstract interpretation: The first work relies on a theorem prover to compute the best abstract transfer function for shape analysis. The second work uses an interactive theorem prover to verify that some program parts satisfy their specification; then, a static analysis that assumes the specifications is run on the whole program to prove its correctness.

Henzinger *et al.* [24,23] use the proof of unsatisfiability produced by a theorem prover to systematically reduce the abstract models checked by the BLAST model checker. The technique used in BLAST has several intriguing similarities to our technique. Two differences are that (i) our analysis is mainly performed inside the theorem prover, whereas BLAST is a separate tool built around a model checker, and (ii) our technique uses widening, whereas BLAST uses Craig's interpolation.

5 Conclusion

We have presented a technique that combines the precision and flexibility of a theorem prover with the power of an abstract interpretation-based static analyzer. The verification conditions are generated from the program source, and they are passed to an automatic theorem prover. The prover tries to prove the verification conditions, and it dynamically invokes the abstract interpreter for the inference of (more precise) loop invariants on a *subset* of the program traces. The abstract interpreter infers a loop invariant that is particular to this set of execution traces, and passes it back to the theorem prover, which then continues the proof. We obtain a program analysis that is a form of trace partitioning. The partitioning is value-based (the partition is done on the values of program variables) and automatic (the theorem prover chooses the partitions automatically).

We have a prototype implementation of our technique, built as an extension of the Zap theorem prover. We have used our prototype to verify the program in our running example, as well as several other small example programs.

We plan to extend our work in several directions. For the implementation, we plan (i) to perform several optimizations in the interaction of the prover and the abstract interpreter, *e.g.*, by caching the calls to the GenerateFacts routine or by a smarter handling of the analysis of nested loops; and (ii) to include our work in the Boogie static program verifier, which is part of the Spec# programming system [5]. This second point will give us easier access to many larger programs. But it will also require some extensions to the theoretical framework presented in this paper, because the starting point of Boogie's inference is a language with basic blocks and goto statements [14,4] (as opposed to the structured source language we have presented here).

We also want to extend our work to handle recursive procedures. In this case, the abstract interpreter will be invoked to generate not only loop invariants, but also procedure summaries, which we are hopeful can be handled in a similar way on demand. If obtained, such an extension of our technique to procedure summaries could provide a new take on interprocedural inference.

It will also be of interest to instantiate the static analyzer with non-numeric abstract domains, for instance a domain for shape analysis (*e.g.*, [35]). In this case, we expect the abstract interpreter to infer invariants on the shape of the heap that are useful to prove heap properties such as that a method correctly manipulates a list (*e.g.*, “if the input is an acyclic list, then the output is also an acyclic list”), and that combined with the inference of class invariants [31,8] may allow the verification of class-level specifications (*e.g.*, “the class implements an acyclic list”).

Acknowledgments. We are grateful to Microsoft Research for hosting Logozzo when we developed the theoretical framework of this work, and to Radhia Cousot for hosting us when we built the prototype implementation. We thank the audiences at the AVIS 2005 workshop and the IFIP Working Group 2.3 meeting in Niagara Falls for providing feedback that has led us to a deeper understanding of the new technique.

References

1. Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *15th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL’88)*, pages 1–11. ACM, January 1988.
2. Thomas Ball, Byron Cook, Shuvendu K. Lahiri, and Lintao Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Computer Aided Verification, 16th International Conference, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 457–461. Springer, July 2004.
3. Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Model Checking Software, 8th International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer, May 2001.
4. Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*. ACM, 2005.
5. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.
6. Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer, July 2004.
7. François Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–423, 1992.
8. Bor-Yuh Evan Chang and K. Rustan M. Leino. Inferring object invariants. In *Proceedings of the First International Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL 2005)*, volume 131 of *Electronic Notes in Theoretical Computer Science*. Elsevier, January 2005.
9. Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
10. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL ’77)*, pages 238–252. ACM, January 1977.
11. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’79)*, pages 269–282. ACM, 1979.

12. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '78)*, pages 84–97. ACM, 1978.
13. Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. In *Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*, May 2002.
14. Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report 2005-70, Microsoft Research, May 2005.
15. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
16. Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In *15th Computer-Aided Verification conference (CAV)*, July 2003.
17. Cormac Flanagan, Rajeev Joshi, and James B. Saxe. An explicating theorem prover for quantified formulas. Technical Report HPL-2004-199, HP Labs, 2004.
18. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *28th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'01)*, pages 193–205, 2001.
19. Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposium in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.
20. Roberto Giacobazzi and Francesco Ranzato. The reduced relative power operation on abstract domains. *Theoretical Computer Science*, 216(1-2):159–211, March 1999.
21. Susanne Graf and Hassen Saïdi. Construction of abstract state graphs via PVS. In *Computer Aided Verification, 9th International Conference (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.
22. Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Static Analysis Symposium (SAS '98)*, volume 1503 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 1998.
23. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of POPL 2004*, pages 232–244. ACM, 2004.
24. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *29th Syposium on Principles of Programming Languages (POPL'02)*, pages 58–70. ACM, 2002.
25. C. A. R. Hoare. An axiomatic approach to computer programming. *Communications of the ACM*, 12:576–580,583, 1969.
26. Bertrand Jeannet, Nicolas Halbwachs, and Pascal Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium (SAS'99)*, volume 1694 of *Lecture Notes in Computer Science*. Springer, September 1999.
27. K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, March 2005.
28. K. Rustan M. Leino, Madan Musuvathi, and Xinming Ou. A two-tier technique for supporting quantifiers in a lazily proof-explicating theorem prover. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005*, volume 3440 of *Lecture Notes in Computer Science*, pages 334–348. Springer, April 2005.
29. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999. Also available as Technical Note 1999-002, Compaq Systems Research Center.
30. Francesco Logozzo. Approximating module semantics with constraints. In *Proceedings of the 19th ACM SIGAPP Symposium on Applied Computing (SAC 2004)*, pages 1490–1495. ACM, March 2004.

31. Francesco Logozzo. *Modular Static Analysis of Object-oriented Languages*. PhD thesis, École Polytechnique, 2004.
32. Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *14th European Symposium on Programming (ESOP 2005)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 2005.
33. Antoine Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.
34. Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.
35. Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
36. Greta Yorsh, Thomas W. Reps, and Shmuel Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, pages 530–545, 2004.
37. Karen Zee, Patrick Lam, Viktor Kuncak, and Martin C. Rinard. Combining theorem proving with static analysis for data structure consistency. In *International Workshop on Software Verification and Validation*, November 2004.

Integrating Physical Systems in the Static Analysis of Embedded Control Software[⋆]

Patrick Cousot

École Normale Supérieure, Paris, France
Patrick.Cousot@ens.fr
www.di.ens.fr/~cousot

Abstract Interpretation

Abstract interpretation is a theory of effective abstraction and/or approximation of discrete mathematical structures as found in the semantics of programming languages, modelling program executions, hence program properties, at various levels of abstraction [3,7,8,10,12].

Static Analysis by Abstract Interpretation

The prominent practical application of abstract interpretation has been to static program analysis, that is the automatic (without any human intervention), static (at compile time) determination of dynamic program properties (that always hold at runtime) involving complex abstractions of the infinite state operational semantics (e.g. [4,5,9,11]). Abstract interpretation fights undecidability and complexity by approximation of the program execution model which may lead to false alarms in correctness proofs. This happens whenever the combination of the abstract domains involved in the analyzer is not precise enough to express any inductive argument necessary in the correctness proof. Hence, among other possible alternatives, the idea to specialize static analyzers to well-defined families of programs and properties for which abstract domains can be designed to express all information necessary to perform inductive proofs [6].

Static Analysis of Embedded Control Software

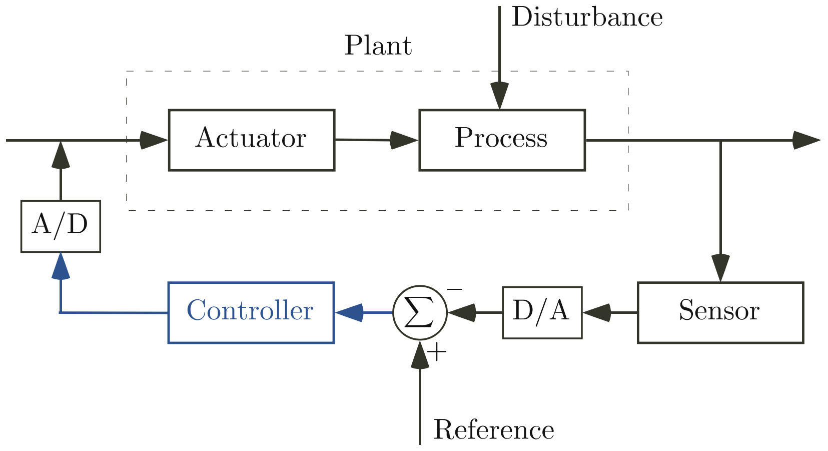
This approach was successfully illustrated by the ASTRÉE static analyzer which is specialized for proving the absence of run-time errors in synchronous, time-triggered, real-time, safety critical, embedded software written or automatically generated in the C programming language [1,2,13]. It was able to prove the absence of run-time errors in large industrial avionic control-command programs [14]. It is a remarkable well-design criterion that the absence of runtime errors can be proved in such control/command software without any hypotheses on the controlled systems (but, maybe, for ranges of variation of very few volatile input variables). This means that the software will go on functioning without any

[⋆] This work was supported in part by the Jerome Clarke Hunsaker Visiting Professorship of the MIT Aeronautics and Astronautics Department in 2005. I thank John Deyst and Éric Féron for stimulating discussions.

runtime error whichever the behavior of the controlled system can be, as long as the processor on which the program is running does not fail (a situation which can be handled by fault-tolerance techniques [15]). Obviously not all desirable properties of the controlled physical system can be proved in this way by a very coarse abstraction of the properties of this physical system.

Integrating Physical Systems in the Static Analysis of Embedded Control Software

To go beyond, e.g. to prove robustness or stability, is is necessary to take into account the full feedback control system that is the controller (from which the control/command program was generated) but also the mathematical model of the physical system (either in the form of differential equations, difference equations or of a numerical model as given e.g. in Simulink$^{\text{TM}}$):



We advocate an approach in which code is generated by discretization both for the continuous dynamic nonlinear model of the controlled system (e.g. from the block diagram description of the plant, actuators and sensors) and for the digital implementation of the controller (as given by the control/command program to be verified).

This code can be that of a specification language when reasoning at the model level or that of a programming language when reasoning at the implementation model.

A static analysis of this code can provide information (like reachability sets) which can hardly be discovered by traditional simulation or test techniques. Such numerical simulations also involve discretization techniques and floating-point computations which might not be the same as those involved in the generated control/command program. Such intricate differences would disappear in an integrated approach.

Taking the environment of execution of the control/command program into account allows for more refined properties of this control/command program to be proved such as reachability in the actual context of use, reactivity, stability, uncertainty and robustness, performance validation, etc of feedback control. Such properties are not always easily expressible as traditional temporal properties commonly used in computer science correctness proofs.

By static analysis, such refined properties can be verified from the more or less idealized and precise model of the controller and plant down to the actual embedded control program. Information can be translated between levels of refinement to ease static analysis or checking at lower levels and to ensure coherence and soundness of the inferred information at all levels of refinement. The verification is thus performed from the model to the derived program with respect to the full specification of the execution environment. A central advantage of this integrated approach is the potential for early discovery of design errors much before the costly experimentations on an actual physical implementation.

Convex Abstractions

We present new abstract interpretations and abstract domains issued from modern control theory and convex optimization as a first step towards reaching these ambitious objectives of integrating physical systems in the static analysis of embedded control software.

References

1. The ASTRÉE Static Analyzer. `www.astree.ens.fr`.
2. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (PLDI'03), San Diego, California , USA, June 7–14, 2003, pp. 196–207. ACM Press, 2003.
3. P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.* Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, 21 Mar. 1978.
4. P. Cousot. Types as abstract interpretations, invited paper. In *24th POPL*, pages 316–331, Paris, Jan. 1997. ACM Press.
5. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, volume 173, pages 421–505. NATO Science Series, Series F: Computer and Systems Sciences. IOS Press, 1999.
6. P. Cousot. Partial completeness of abstract fixpoint checking, invited paper. In B. Choueiry and T. Walsh, editors, *Proc. 4th Int. Symp. SARA '2000*, Horseshoe Bay, LNAI 1864, pages 1–25. Springer, 26–29 Jul. 2000.
7. P. Cousot. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, pages 1–24, Paris, France, January 17-19, 2005. Lecture Notes in Computer Science, volume 3385, Springer, Berlin.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252, Los Angeles, 1977. ACM Press.

9. P. Cousot and R. Cousot. Static determination of dynamic properties of generalized type unions. In *ACM Symposium on Language Design for Reliable Software*, Raleigh, ACM SIGPLAN Not. 12(3):77–94, 1977.
10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th POPL*, pages 269–282, San Antonio, 1979. ACM Press.
11. P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In A. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Macmillan, 1984.
12. P. Cousot and R. Cousot. Basic concepts of abstract interpretation. In P. Jacquart, editor, *Building the Information Society*, chapter 4, pages 359–366. Kluwer Acad. Pub., 2004.
13. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyser. In M. Sagiv, editor, *Proc. 14th ESOP '2005, Edinburg, UK*, volume 3444 of *LNCS*, pages 21–30. Springer, Apr. 2-10, 2005.
14. J. Souyris. Industrial experience of abstract interpretation-based static analyzers. In P. Jacquart, editor, *Building the Information Society*, chapter 4, pages 393–400. Kluwer Acad. Pub., 2004.
15. P. Traverse, I. Lacaze, and J. Souyris. Airbus ly-by-wire — a total approach to dependability. In P. Jacquart, editor, *Building the Information Society*, chapter 3, pages 191–212. Kluwer Acad. Pub., 2004.

Reflection Analysis for Java

Benjamin Livshits, John Whaley, and Monica S. Lam[⋆]

Computer Science Department, Stanford University,
Stanford, CA 94305, USA
{livshits, jwhaley, lam}@cs.stanford.edu

Abstract. Reflection has always been a thorn in the side of Java static analysis tools. Without a full treatment of reflection, static analysis tools are both *incomplete* because some parts of the program may not be included in the application call graph, and *unsound* because the static analysis does not take into account reflective features of Java that allow writes to object fields and method invocations. However, accurately analyzing reflection has always been difficult, leading to most static analysis tools treating reflection in an unsound manner or just ignoring it entirely. This is unsatisfactory as many modern Java applications make significant use of reflection.

In this paper we propose a static analysis algorithm that uses points-to information to approximate the targets of reflective calls as part of call graph construction. Because reflective calls may rely on input to the application, in addition to performing reflection resolution, our algorithm also discovers all places in the program where user-provided specifications are necessary to fully resolve reflective targets. As an alternative to user-provided specifications, we also propose a reflection resolution approach based on type cast information that reduces the need for user input, but typically results in a less precise call graph.

We have implemented the reflection resolution algorithms described in this paper and applied them to a set of six large, widely-used benchmark applications consisting of more than 600,000 lines of code combined. Experiments show that our technique is effective for resolving most reflective calls without any user input. Certain reflective calls, however, cannot be resolved at compile time precisely. Relying on a user-provided specification to obtain a conservative call graph results in graphs that contain 1.43 to 6.58 times more methods that the original. In one case, a conservative call graph has 7,047 more methods than a call graph that does not interpret reflective calls. In contrast, ignoring reflection leads to missing substantial portions of the application call graph.

1 Introduction

Whole-program static analysis requires knowing the targets of function or method calls. The task of computing a program's call graph is complicated for a language like Java because of virtual method invocations and reflection. Past research has addressed the analysis of function pointers in C as well as virtual method calls in C++ and Java. Reflection, however, has mostly been neglected.

[⋆] This work was supported by NSF Grant No. 0326227 and an Intel Graduate Fellowship.

Reflection in Java allows the developer to perform runtime actions given the descriptions of the objects involved: one can create objects given their class names, call methods by their name, and access object fields given their name [1]. Because names of methods to be invoked can be supplied by the user, especially in the presence of dynamic class loading, precise static construction of a call graph is generally undecidable. Even if we assume that all classes that may be used are available for analysis, without placing *any restrictions* of the targets of reflective calls, a sound (or conservative) call graph would be prohibitively large.

Many projects that use static analysis for optimization, error detection, and other purposes ignore the use of reflection, which makes static analysis tools *incomplete* because some parts of the program may not be included in the call graph and potentially *unsound*, because some operations, such as reflectively invoking a method or setting an object field, are ignored. Our research is motivated by the practical need to improve the coverage of static error detection tools [2,3,4]. The success of such tools in Java is predicated upon having a call graph available to the error detection tool. Unless reflective calls are interpreted, the tools run the danger of only analyzing a small portion of the available code and giving the developer a false sense of security when no bugs are reported. Moreover, when static results are used to reduce runtime instrumentation, all parts of the application that are used at runtime *must* be statically analyzed.

A recent paper by Hirzel, Diwan, and Hind proposes the use of dynamic instrumentation to collect the reflection targets discovered at run time [5]. They use this information to extend Andersen's context-insensitive, inclusion-based pointer analysis for Java into an online algorithm [6]. Reflective calls are generally used to offer a choice in the application control flow, and a dynamic application run typically includes only several of all the possibilities. However, analyses used for static error detection and optimization often require a *full* call graph of the program in order to achieve complete coverage.

In this paper we present a static analysis algorithm that uses points-to information to determine the targets of reflective calls. Often the targets of reflective calls can be determined precisely by analyzing the flow of strings that represent class names throughout the program. This allows us to precisely resolve many reflective calls and add them to the call graph. However, in some cases reflective call targets may depend on user input and require user-provided specifications for the call graph to be determined. Our algorithm determines all *specification points* — places in the program where user-provided specification is needed to determine reflective targets. The user is given the option to provide a specification and our call graph is complete with respect to the specifications provided [7].

Because providing reflection specifications can be time-consuming and error-prone, we also provide a conservative, albeit sometimes imprecise, approximation of targets of reflective calls by analyzing how type casts are used in the program. A common coding idiom consists of casting the result of a call to `Class.newInstance` used to create new objects to a more specific type before the returned object can be used. Relying on cast information allows us to produce a conservative call graph approximation without requiring user-provided reflection specifications in most cases. A flow diagram summarizing the stages of our analysis is shown in Figure 1.

Our reflection resolution approach hinges on three assumptions about the use of reflection: (a) all the class files that may be accessed at runtime are

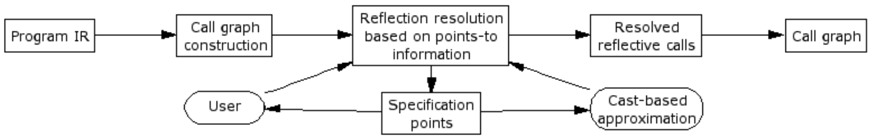


Fig. 1. Architecture of our static analysis framework

available for analysis; (b) the behavior of `Class.forName` is consistent with its API definition in that it returns a class whose name is specified by the first parameter, and (c) cast operations that operate on the results of `Class.newInstance` calls are correct. In rare cases when no cast information is available to aid with reflection resolution, we report this back to the user as a situation requiring specification.

1.1 Contributions

This paper makes the following contributions:

– We formulate a set of natural assumptions that hold in most Java applications and make the use of reflection amenable to static analysis.
– We propose a call graph construction algorithm that uses points-to information about strings used in reflective calls to statically find potential call targets. When reflective calls cannot be fully "resolved" at compile time, our algorithms determines a set of specification points — places in the program that require user-provided specification to resolve reflective calls.
– As an alternative to having to provide a reflection specification, we propose an algorithm that uses information about type casts in the program to statically approximate potential targets of reflective calls.
– We provide an extensive experimental evaluation of our analysis approach based on points-to results by applying it to a suite of six large open-source Java applications consisting of more than 600,000 lines of code combined. We evaluate how the points-to and cast-based analyses of reflective calls compare to a local intra-method approach. While all these analyses find at least one constant target for most `Class.forName` call sites, they only moderately increase the call graph size. However, the conservative call graph obtained with the help of a user-provided specification results is a call graph than is almost 7 times as big as the original. We assess the amount of effort required to come up with a specification and how cast-based information can significantly reduce the specification burden placed on the user.

1.2 Paper Organization

The rest of the paper is organized as follows. In Section 2, we provide background information about the use of reflection in Java. In Section 3, we lay out the simplifying assumptions made by our static analysis. In Sections 4 we describe our analysis approach. Section 5 provides a comprehensive experimental evaluation. Finally, in Sections 6 and 7 we describe related work and conclude.

2 Overview of Reflection in Java

In this section we first informally introduce the reflection APIs in Java. The most typical use of reflection by far is for creating new objects given the object class name. The most common usage idiom for reflectively creating an object is shown in Figure 2. Reflective APIs in Java are used for object creation, method invocation, and field access, as described below. Because of the space limitations, in this section we only briefly outline the relevant reflective APIs. Interested readers are encouraged to refer to our technical report for a complete treatment and a case study of reflection uses in our benchmarks applications [8].

```
String className = ...;
Class c  = Class.forName(className);
Object o = c.newInstance();
T      t = (T) o;
```

Fig. 2. Typical use of reflection to create new objects

Object Creation. Object creation APIs in Java provide a way to programmatically create objects of a class, whose name is provided at runtime; parameters of the object constructor can be passed in as necessary. Obtaining a class given its name is most typically done using a call to one of the static functions `Class.forName(String, ...)` and passing the class name as the first parameter. We should point out that while `Class.forName` is the most common way to obtain a class given its name, it may not be the only method for doing so. An application may define a native method that implements the same functionality. The same observation applies to other standard reflective API methods.

The commonly used Java idiom `T.class`, where `T` is a class is translated by the compiler to a call to `Class.forName(T.getName())`. Since our reflection resolution algorithm works at the byte code level, `T.class` constructs do not require a special treatment. Creating an object with an empty constructor is achieved through a call to `newInstance` on the appropriate `java.lang.Class` object, which provides a runtime representation of a class.

Method Invocation. Methods are obtained from a `Class` object by supplying the method signature or by iterating through the array of `Methods` returned by one of `Class` functions. `Methods` are subsequently invoked by calling `Method.invoke`.

Accessing Fields. Fields of Java runtime objects can be read and written at runtime. Calls to `Field.get` and `Field.set` can be used to get and set fields containing objects. Additional methods are provided for fields of primitive types.

3 Assumptions About Reflection

This section presents assumptions we make in our static analysis for resolving reflection in Java programs. We believe that these assumptions are quite reasonable and hold for many real-life Java applications.

The problem of precisely determining the classes that an application may access is undecidable. Furthermore, for applications that access the network, the set of classes that may be accessed is *unbounded*: we cannot possibly hope to analyze all classes that the application may conceivably download from the net and load at runtime. Programs can also dynamically generate classes to be subsequently loaded. Our analysis assumes a closed world, as defined below.

Assumption 1. Closed world.
We assume that only classes reachable from the class path at analysis time can be used by the application at runtime.

In the presence of user-defined class loaders, it is impossible to statically determine the behavior of function `Class.forName`. If custom class loaders are used, the behavior of `Class.forName` can change; it is even possible for a malicious class loader to return completely unrelated classes in response to a `Class.forName` call. The following assumption allows us to interpret calls to `Class.forName`.

Assumption 2. Well-behaved class loaders.
The name of the class returned by a call to `Class.forName(className)` *equals* `className`*.*

To check the validity of Assumption 2, we have instrumented large applications to observe the behavior of `Class.forName`; we have never encountered a violation of this assumption. Finally, we introduce the following assumption that allows us to leverage type cast information contained in the program to constrain the targets of reflective calls.

Assumption 3. Correct casts.
Type cast operations that always operate on the result of a call to `newInstance` *are correct; they will always succeed without throwing a* `ClassCastException`*.*

We believe this to be a valid practical assumption: while it is possible to have casts that fail, causing an exception that is caught so that the instantiated object can be used afterwards, we have not seen such cases in practice. Typical `catch` blocks around such casts lead to the program terminating with an error message.

4 Analysis of Reflection

In this section, we present techniques for resolving reflective calls in a program. Our analysis consists of the following three steps:

1. We use a sound points-to analysis to determine all the possible sources of strings that are used as class names. Such sources can either be constant strings or derived from external sources. The pointer analysis-based approach *fully resolves* the targets of a reflective call if constant strings account for all the possible sources. We say that a call is *partially resolved* if the sources can be either constants or inputs and *unresolved* if the sources can only be inputs. Knowing which external sources may be used as class names is useful because users can potentially specify all the possible values; typical examples are return results of file read operations. We refer to program points where the input strings are defined as *specification points.*

2. Unfortunately the number of specification points in a program can be large. Instead of asking users to specify the values of every possible input string, our second technique takes advantage of casts, whenever available, to determine a conservative approximation of targets of reflective calls *that are not fully resolved*. For example, as shown in Figure 2, the call to `Class.newInstance`, which returns an `Object`, is always followed by a cast to the appropriate type before the newly created object can be used. Assuming no exception is raised, we can conclude that the new object must be a subtype of the type used in the cast, thus restricting the set of objects that may be instantiated.
3. Finally, we rely on user-provided specification for the remaining set of calls — namely calls whose source strings are not all constants — in order to obtain a conservative approximation of the call graph.

We start by describing the call graph discovery algorithm in Section 4.1 as well as how reflection resolution fits in with call graph discovery. Section 4.2 presents a reflection resolution algorithm based on pointer analysis results. Finally, Section 4.3 describes our algorithm that leverages type cast information for conservative call graph construction without relying on user-provided specifications.

4.1 Call Graph Discovery

Our static techniques to discover reflective targets are integrated into a context-insensitive points-to analysis that discovers the call graph on the fly [9]. As the points-to analysis finds the pointees of variables, type information of these pointees is used to resolve the targets of virtual method invocations, increasing the size of the call graph, which in turn is used to find more pointees. Our analysis of reflective calls further expands the call graph, which is used in the analysis to generate more points-to relations, leading to bigger call graphs. The discovery algorithm terminates when a fixpoint is reached and no more call targets or points-to relations can be found.

By using a points-to analysis to discover the call graph, we can obtain a more accurate call graph than by using a less precise technique such as class hierarchy analysis CHA [10] or rapid type analysis RTA [11]. We use a context-insensitive version of the analysis because context sensitivity does not seem to substantially improve the accuracy of the call graph [9,12].

4.2 Pointer Analysis for Reflection

This section describes how we leverage pointer analysis results to resolve calls to `Class.forName` and track `Class` objects. This can be used to discover the types of objects that can be created at calls to `Class.newInstance`, along with resolving reflective method invocations and field access operations. Pointer analysis is also used to find specification points: external sources that propagate string values to the first argument of `Class.forName`.

Reflection and Points-to Information. The programming idiom that motivated the use of points-to analysis for resolving reflection was first presented in Figure 2. This idiom consists of the following steps:

1. Obtain the name of the class for the object that needs to be created.
2. Create a `Class` object by calling the static method `Class.forName`.
3. Create the new object with a call to `Class.newInstance`.
4. Cast the result of the call to `Class.newInstance` to the necessary type in order to use the newly created object.

When interpreting this idiom statically, we would like to "resolve" the call to `Class.newInstance` in step 3 as a call to the default constructor `T()`. However, analyzing even this relatively simple idiom is nontrivial.

The four steps shown above can be widely separated in the code and reside in different methods, classes, or jar libraries. The `Class` object obtained in step 2 may be passed through several levels of function calls before being used in step 3. Furthermore, the `Class` object can be deposited in a collection to be later retrieved in step 3. The same is true for the name of the class created in step 1 and used later in step 2. To determine how variables `className`, `c`, `o`, and `t` defined and used in steps 1–4 may be related, we need to know what runtime objects they may be referring to: a problem addressed by *points-to* analysis. Point-to analysis computes which objects each program variable may refer to.

Resolution of `Class.newInstance` of `Class.forName` calls is not the only thing made possible with points-to results: using points-to analysis, we also track `Method`, `Field`, and `Constructor` objects. This allows us to correctly resolve reflective method invocations and field accesses. Reflection is also commonly used to invoke the class constructor of a given class via calling `Class.forName` with the class name as the first argument. We use points-to information to determine potential targets of `Class.forName` calls and add calls to class constructors of the appropriate classes to the call graph.

The bddbddb Program Database. In the remainder of this section we describe how pointer information is used for reflection resolution. We start by describing how the input program can be represented as a set of relations in bddbddb, a BDD-based program database [9,13]. The program database and the associated constraint resolution tool allows program analyses to be expressed in a succinct and natural fashion as a set of rules in Datalog, a logic programming language. Points-to information is compactly represented in bddbddb with binary decision diagrams (BDDs), and can be accessed and manipulated efficiently with Datalog queries. The program representation as well as pointer analysis results are stored as relations in the bddbddb database. The domains in the database include invocation sites I, variables V, methods M, heap objects named by their allocation site H, types T, and integers Z.

The source program is represented as a number of input relations. For instance, relations *actual* and *ret* represent parameter passing and method returns, respectively. In the following, we say that predicate $A(x_1, \ldots, x_n)$ is true if tuple $(x_1, \ldots, x_n)$ is in relation A. Below we show the definitions of Datalog relations used to represent the input program:

actual: $I \times Z \times V$. *actual*(i, z, v) means that variable v is zth argument of the method call at i.

ret: $I \times V$. *ret*(i, v), means that variable v is the return result of the method call at i.

assign: $V \times V$. *assign*(v_1, v_2) means that there is an implicit or explicit assignment statement $v_1 = v_2$ in the program.

load: $V \times F \times V$. *load*(v_1, f, v_2) means that there is a load statement $v_2 = v_1.f$ in the program.

store: $V \times F \times V$. *store*(v_1, f, v_2) means that there is a store statement $v_1.f = v_2$ in the program.

string2class: $H \times T$. *string2class*(s, t) means that string constant s is the string representation of the name of type t.

calls: $I \times M$ is the invocation relation. *calls*(i, m) means that invocation site i may invoke method m.

Points-to results are represented with the relation *vP*:

vP: $V \times H$ is the variable points-to relation. *vP*(v, h) means that variable v may point to heap object h.

A Datalog query consists of a set of rules, written in a Prolog-style notation, where a predicate is defined as a conjunction of other predicates. For example, the Datalog rule $D(w, z) :- A(w, x), B(x, y), C(y, z)$. says that "$D(w, z)$ is true if $A(w, x)$, $B(x, y)$, and $C(y, z)$ are all true."

Reflection Resolution Algorithm. The algorithm for computing targets of reflective calls is naturally expressed in terms of Datalog queries. Below we define Datalog rules to resolve targets of `Class.newInstance` and `Class.forName` calls. Handling of constructors, methods, and fields proceed similarly.

To compute reflective targets of calls to `Class.newInstance`, we define two Datalog relations. Relation *classObjects* contains pairs $\langle i, t \rangle$ of invocations sites $i \in I$ calling `Class.forName` and types $t \in T$ that may be returned from the call. We define *classObjects* using the following Datalog rule:

$$\begin{aligned} \mathit{classObjects}(i,t)\ :-\ & \mathit{calls}(i, \text{“}\texttt{Class.forName}\text{”}),\\ & \mathit{actual}(i,1,v), \mathit{vP}(v,s), \mathit{string2class}(s,t). \end{aligned}$$

The Datalog rule for *classObjects* reads as follows. Invocation site i returns an object of type t if the call graph relation *calls* contains an edge from i to "`Class.forName`", parameter 1 of i is v, v points to s, and s is a string that represents the name of type t.

Relation *newInstanceTargets* contains pairs $\langle i, t \rangle$ of invocation sites $i \in I$ calling `Class.newInstance` and classes $t \in T$ that may be reflectively invoked by the call. The Datalog rule to compute *newInstanceTargets* is:

$$\begin{aligned} \mathit{newInstanceTargets}(i,t)\ :-\ & \mathit{calls}(i, \text{“}\texttt{Class.newInstance}\text{”}),\\ & \mathit{actual}(i,0,v), \mathit{vP}(v,c),\\ & \mathit{vP}(v_c,c), \mathit{ret}(i_c,v_c), \mathit{classObjects}(i_c,t). \end{aligned}$$

The rule reads as follows. Invocation site i returns a new object of type t if the call graph relation *calls* contains an edge from i to `Class.newInstance`, parameter 0 of i is v, v is aliased to a variable v_c that is the return value of invocation site i_c, and i_c returns type t. Targets of `Class.forName` calls are resolved and calls to the appropriate class constructors are added to the invocation relation *calls*:

$$\mathit{calls}(i,m)\ :-\ \mathit{classObjects}(i,t), m = t + \text{“}. < \texttt{clinit} >\text{”}.$$

(The "+" sign indicates string concatenation.) Similarly, having computed relation *newInstanceTargets*(i, t), we add these reflective call targets invoking the appropriate type constructor to the call graph relation *calls* with the rule below:

```
loadImpl() @ 43 InetAddress.java:1231       => java.net.Inet4AddressImpl
loadImpl() @ 43 InetAddress.java:1231       => java.net.Inet6AddressImpl
...
lookup() @ 86 AbstractCharsetProvider.java:126 => sun.nio.cs.ISO_8859_15
lookup() @ 86 AbstractCharsetProvider.java:126 => sun.nio.cs.MS1251
...
tryToLoadClass() @ 29 DataFlavor.java:64       => java.io.InputStream
...
```

Fig. 3. A fragment of a specification file accepted by our system. A string identifying a call site to `Class.forName` is mapped to a class name that that call may resolve to.

$$\mathit{calls}(i, m) \;:-\; \mathit{newInstanceTargets}(i, t), m = t + \text{“}.<\texttt{init}>\text{”}.$$

Handling `Constructor` and Other Objects. Another technique of reflective object creation is to use `Class.getConstructor` to get a `Constructor` object, and then calling `newInstance` on that. We define a relation *constructorTypes* that contains pairs $\langle i, t\rangle$ of invocations sites $i \in I$ calling `Class.getConstructor` and types $t \in T$ of the type of the constructor:

$$\begin{aligned}\mathit{constructorTypes}(i, t) \;:-\; & \mathit{calls}(i, \text{“}\texttt{Class.getConstructor}\text{”}),\\ & \mathit{actual}(i, 0, v), \mathit{vP}(v, h), \mathit{classObjects}(h, t).\end{aligned}$$

Once we have computed *constructorTypes*, we can compute more *newInstanceTargets* as follows:

$$\begin{aligned}\mathit{newInstanceTargets}(i, t) \;:-\; & \mathit{calls}(i, \text{“}\texttt{Class.newInstance}\text{”}),\\ & \mathit{actual}(i, 0, v), \mathit{vP}(v, c), \mathit{vP}(v_c, c), \mathit{ret}(i_c, v_c),\\ & \mathit{constructorTypes}(i_c, t).\end{aligned}$$

This rule says that invocation site i calling “`Class.newInstance`” returns an object of type t if parameter 0 of i is v, v is aliased to the return value of invocation i_c which calls “`Class.getConstructor`”, and the call to i_c is on type t.

In a similar manner, we can add support for `Class.getConstructors`, along with support for reflective field, and method accesses. The specification of these are straightforward and we do not describe them here. Our actual implementation completely models all methods in the Java Reflection API. We refer the reader to a technical report we have for more details [8].

Specification Points and User-Provided Specifications. Using a points-to analysis also allows us to determine, when a non-constant string is passed to a call to `Class.forName`, the *provenance* of that string. The *provenance* of a string is in essence a backward data slice showing the flow of data to that string. Provenance allows us to compute *specification points*—places in the program where external sources are read by the program from a configuration file, system properties, etc. For each specification point, the user can provide values that may be passed into the application.

We compute the provenance by propagating through the assignment relation *assign*, aliased loads and stores, and string operations. To make the specification points as close to external sources as possible, we perform a simple analysis of strings to do backward propagation through string concatenation operations. For brevity, we only list the `StringBuffer.append` method used by the Java compiler to expand string concatenation operations here; other string operations work in a similar manner. The following rules for relation *leadsToForName* detail provenance propagation:

$$\begin{aligned}
&\mathit{leadsToForName}(v,i) &&:- \mathit{calls}(i, \text{``}\texttt{Class.forName}\text{''}), \mathit{actual}(i,1,v).\\
&\mathit{leadsToForName}(v_2,i) &&:- \mathit{leadsToForName}(v_1,i), \mathit{assign}(v_1,v_2).\\
&\mathit{leadsToForName}(v_2,i) &&:- \mathit{leadsToForName}(v_1,i),\\
&&&\quad \mathit{load}(v_3,f,v_1), \mathit{vP}(v_3,h_3), \mathit{vP}(v_4,h_3), \mathit{store}(v_4,f,v_2).\\
&\mathit{leadsToForName}(v_2,i) &&:- \mathit{leadsToForName}(v_1,i), \mathit{ret}(i_2,v_1),\\
&&&\quad \mathit{calls}(i_2, \text{``}\texttt{StringBuffer.append}\text{''}), \mathit{actual}(i_2,0,v_2).\\
&\mathit{leadsToForName}(v_2,i) &&:- \mathit{leadsToForName}(v_1,i), \mathit{ret}(i_2,v_1),\\
&&&\quad \mathit{calls}(i_2, \text{``}\texttt{StringBuffer.append}\text{''}), \mathit{actual}(i_2,1,v_2).\\
&\mathit{leadsToForName}(v_2,i) &&:- \mathit{leadsToForName}(v_1,i), \mathit{actual}(i_2,0,v_1),\\
&&&\quad \mathit{calls}(i_2, \text{``}\texttt{StringBuffer.append}\text{''}), \mathit{actual}(i_2,1,v_2).
\end{aligned}$$

To compute the specification points necessary to resolve `Class.forName` calls, we find endpoints of the *leadsToForName* propagation chains that are *not* string constants that represent class names. These will often terminate in the return result of a call to `System.getProperty` in the case of reading from a system property or `BufferedReader.readLine` in the case of reading from a file. By specifying the possible values at that point that are appropriate for the application being analyzed, the user can construct a complete call graph.

Our implementation accepts specification files that contain a simple textual map of a specification point to the constant strings it can generate. A specification point is represented by a method name, bytecode offset, and the relevant line number. An example of a specification file is shown in Figure 3.

4.3 Reflection Resolution Using Casts

For some applications, the task of providing reflection specifications may be too heavy a burden. Fortunately, we can leverage the type cast information present in the program to automatically determine a conservative approximation of possible reflective targets. Consider, for instance, the following typical code snippet:

```
1.   Object o = c.newInstance();
2.   String s = (String) o;
```

The cast in statement 2 *post-dominates* the call to `Class.newInstance` in statement 1. This implies that all execution paths that pass through the call to `Class.newInstance` must also go through the cast in statement 2 [14]. For statement 2 not to produce a runtime exception, `o` must be a subclass of `String`. Thus, only subtypes of `String` can be created as a result of the call to `newInstance`. More generally, if the result of a `newInstance` call is *always* cast to type t, we say that only subtypes of t can be instantiated at the call to `newInstance`.

Relying on cast operations can possibly be unsound as the cast may fail, in which case, the code will throw a `ClassCastException`. Thus, in order to work, our cast-based technique relies on Assumption 3, the correctness of cast operations.

```
1.      UniqueVector voiceDirectories = new UniqueVector();
2.      for (int i = 0; i < voiceDirectoryNames.size(); i++) {
3.          Class c = Class.forName((String) voiceDirectoryNames.get(i),
4.                                               true, classLoader);
5.          voiceDirectories.add(c.newInstance());
6.      }
7.
8.      return (VoiceDirectory[]) voiceDirectories.toArray(new
9.                      VoiceDirectory[voiceDirectories.size()]);
```

Fig. 4. A case in `freetts` where our analysis is unable to determine the type of objects instantiated on line 5 using casts.

Preparing Subtype Information. We rely on the closed world Assumption 2 described in Section 3 to find the set of all classes possibly used by the application. The classes available at analysis time are generally distributed with the application. However, occasionally, there are classes that are generated when the application is compiled or deployed, typically with the help of an Ant script. Therefore, we generate the set of possible classes *after* deploying the application.

We pre-process all resulting classes to compute the subtyping relation $subtype(t_1, t_2)$ that determines when t_1 is a subtype of t_2. Preprocessing even the smallest applications involved looking at many thousands of classes because we consider all the default jars that the Java runtime system has access to. We run this preprocessing step off-line and store the results for easy access.

Using Cast Information. We integrate the information about cast operations directly into the system of constraints expressed in Datalog. We use a Datalog relation *subtype* described above, a relation *cast* that holds the cast operations, and a relation *unresolved* that holds the unresolved calls to `Class.forName`. The following Datalog rule uses cast operations applied to the return result v_{ret} of a call i to `Class.newInstance` to constrain the possible types t_c of `Class` objects c returned from calls sites i_c of `Class.forName`:

$$\begin{aligned} classObjects(i_c, t) \;:-\; & calls(i, \text{“}\texttt{Class.newInstance}\text{”}), actual(i, 0, v), vP(v, c), \\ & ret(i, v_{ret}), cast(\_, t_c, v_{ret}), subtype(t, t_c), \\ & unresolved(i_c), vP(v_c, c), ret(i_c, v_c). \end{aligned}$$

Information propagates both forward and backward—for example, casting the result of a call to `Class.newInstance` constrains the `Class` object it is called upon. If the same `Class` object is used in another part of the program, the type constraint derived from the cast will be obeyed.

Problems with Using Casts. Casts are sometimes inadequate for resolving calls to `Class.newInstance` for the following reasons. First, the cast-based approach is inherently imprecise because programs often cast the result of `Class.newInstance` to a very wide type such as `java.io.Serializable`. This produces a lot of *potential* subclasses, only some of which are relevant in practice. Second, as our experiments show, not all calls to `Class.newInstance` have post-dominating casts, as illustrated by the following example.

Example 1. As shown in Figure 4, one of our benchmark applications, `freetts`, places the object returned by `Class.newInstance` into a vector `voiceDirectories` (line 5). Despite the fact that the objects are subsequently cast to type `VoiceDirectory[]` on line 8, intraprocedural post-dominance is not powerful enough to take this cast into account. □

Using cast information significantly reduces the need for user-provided specification in practice. While the version of the analysis that does not use cast information can be made fully sound with user specification as well, we chose to only provide a specification for the cast-based version.

5 Experimental Results

In this section we present a comprehensive experimental evaluation of the static analysis approaches presented in Section 4. In Section 5.1 we describe our experimental setup. Section 5.2 presents an overview our experimental results. Section 5.3 presents our baseline local reflection analysis. In Sections 5.4 and 5.5 we discuss the effectiveness of using the points-to and cast-based reflection resolution approaches, respectively. Section 5.6 describes the specifications needed to obtain a sound call graph approximation. Section 5.7 compares the overall sizes of the call graph for the different analysis versions presented in this section.

5.1 Experimental Setup

We performed our experiments on a suite of six large, widely-used open-source Java benchmark applications. These applications were selected among the most popular Java projects available on SourceForge. We believe that real-life applications like these are more representative of how programmers use reflection than synthetically created test suites, or SPEC JVM benchmarks, most of which avoid reflection altogether.

Summary of information about the applications is provided in Figure 5. Notice that the traditional lines of code size metric is somewhat misleading in the case of applications that rely on large libraries. Many of these benchmarks depend of massive libraries, so, while the application code may be small, the full size of the application executed at runtime is quite large. The last column of the

Benchmark	Description	Line count	File count	Jars	Available classes
`jgap`	genetic algorithms package	32,961	172	9	62,727
`freetts`	speech synthesis system	42,993	167	19	62,821
`gruntspud`	graphical CVS client	80,138	378	10	63,847
`jedit`	graphical text editor	144,496	427	1	62,910
`columba`	graphical email client	149,044	1,170	35	53,689
`jfreechart`	chart drawing library	193,396	707	6	62,885
Total		**643,028**	**3,021**	**80**	**368,879**

Fig. 5. Summary of information about our benchmarks. Applications are sorted by the number of lines of code in column 3.

table in Figure 5 lists the number of classes available by the time each application is deployed, including those in the JDK.

We ran all of our experiments on an Opteron 150 machine equipped with 4GB or memory running Linux. JDK version 1.4.2_08 was used. All of the running times for our preliminary implementation were in tens of minutes, which, although a little high, is acceptable for programs of this size. Creating subtype information for use with cast-based analysis took well under a minute.

5.2 Evaluation Approach

We have implemented five different variations of our algorithms: None, Local, Points-to, Casts, and Sound and applied them to the benchmarks described above. None is the base version that performs no reflection resolution; Local performs a simple local analysis, as described in Section 5.3. Points-to and Casts are described in Sections 4.2 and 4.3, respectively.

Version Sound is augmented with a user-provided specification to make the answer conservative. We should point out that only the Sound version provides results that are fully sound: None essentially assumes that reflective calls have no targets. Local only handles reflective calls that can be fully resolved within a single method. Points-to and Casts only provide targets for reflective calls for which either string or cast information constraining the possible targets is available and unsoundly assumes that the rest of the calls have no targets.

Figure 6 summarizes the results of resolving `Class.forName` using all five analysis versions. `Class.forName` calls represent by far the most common kind of reflective operations and we focus on them in our experimental evaluation. To reiterate the definitions in Section 4, we distinguish between:

– *fully resolved calls* to `Class.forName` for which all potential targets are class name constants,
– *partially resolved calls*, which have at least one class name string constant propagating to them, and
– *unresolved calls*, which have no class name string constants propagating to them, only non-constant external sources requiring a specification.

The columns subdivide the total number of calls (T) into fully resolved calls (FR), partially resolved (PR), and unresolved (UR) calls. In the case of Local analysis, there are no partially resolved calls — calls are either fully resolved to constant strings or unresolved. Similarly, in the case of Sound analysis, all calls are either fully resolved or unresolved, as further explained in Section 5.5.

5.3 Local Analysis for Reflection Resolution (Local)

To provide a baseline for comparison, we implemented a local intra-method analysis that identifies string constants passed to `Class.forName`. This analysis catches only those reflective calls that can be resolved completely within a single method. Because this technique does not use interprocedural points-to results, it cannot be used for identification of specification points. Furthermore, because for method invocations and field accesses the names of the method or field are typically *not* locally defined constants, we do not perform resolution of method calls and field accesses in Local.

	None	Local			Points-to				Casts				Sound		
Benchmark	**T**	**T**	**FR**	**UR**	**T**	**FR**	**PR**	**UR**	**T**	**FR**	**PR**	**UR**	**T**	**FR**	**UR**
`jgap`	27	27	19	8	28	20	1	7	28	20	4	4	89	85	4
`freetts`	30	30	21	9	30	21	0	9	34	25	4	5	81	75	6
`gruntspud`	139	139	112	27	142	115	5	22	232	191	19	22	220	208	12
`jedit`	156	156	137	19	161	142	3	16	178	159	12	7	210	197	12
`columba`	104	105	89	16	105	89	2	14	118	101	10	7	173	167	6
`jfreechart`	104	104	91	13	104	91	1	12	149	124	10	15	169	165	4

Fig. 6. Results of resolving `Class.forName` calls for different analysis versions

A significant percentage of `Class.forName` calls can be fully resolved by local analysis, as demonstrated by the numbers in column 4, Figure 6. This is partly due to the fact that it is actually quite common to call `Class.forName` with a constant string parameter for side-effects of the call, because doing so invokes the class constructor. Another common idiom contributing the number of calls resolved by local analysis is `T.class`, which is converted to a call to `Class.forName` and is *always* statically resolved.

5.4 Points-To Information for Reflection Resolution (Points-to)

Points-to information is used to find targets of reflective calls to `Class.forName`, `Class.newInstance`, `Method.invoke`, etc. As can be seen from Figure 6, for all of the benchmarks, Points-to information results in more resolved `Class.forName` calls and fewer unresolved ones compared to Local.

Specification Points. Quite frequently, some sort of specification is required for reflective calls to be fully resolved. Points-to information allows us to provide the user with a list of specification points where inputs needs to be specified for a conservative answer to be obtained. Among the specification points we have encountered in our experiments, calls to `System.getProperty` to retrieve a system variable and calls to `BufferedReader.readLine` to read a line from a file are quite common. Below we provide a typical example of providing a specification.

Example 2. This example describes resolving reflective targets of a call to `Class.newInstance` in `javax.xml.transform.FactoryFinder` in the JDK in order to illustrate the power and limitation of using points-to information. Class `FactoryFinder` has a method `Class.newInstance` shown in Figure 7. The call to `Class.newInstance` occurs on line 9. However, the exact class instantiated at runtime depends on the `className` parameter, which is passed into this function. This function is invoked from a variety of places with the `className` parameter being read from initialization properties files, the console, etc. In only one case, when `Class.newInstance` is called from another function `find` located in another file, is the `className` parameter a string constant.

This example makes the power of using points-to information apparent — the `Class.newInstance` target corresponding to the string constant is often difficult to find by just looking at the code. The relevant string constant was passed down through several levels of method calls located in a different file; it took us more that five minutes of exploration with a powerful code browsing tool to find this case in the source. Resolving this `Class.newInstance` call also requires the user to provide input for four specification points: along with a constant class name,

```
private static Object newInstance(String className,
                ClassLoader classLoader) throws ConfigurationError {
   try {
       Class spiClass;
       if (classLoader == null) {
           spiClass = Class.forName(className);
       }
       ...
       return spiClass.newInstance();
   } catch (...)
       ...
   }
```

Fig. 7. Reflection resolution using points-to results in `javax.xml.transform.FactoryFinder` in the JDK

our analysis identifies two specification points, which correspond to file reads, one access of system properties, and another read from a hash table. □

In most cases, the majority of calls to `Class.forName` are fully resolved. However, a small number of unresolved calls are potentially responsible for a large number of specification points the user has to provide. For Points-to, the average number of specification points per invocation site ranges from 3 for `freetts` to 9 for `gruntspud`. However, for `jedit`, the average number of specification points is 422. Specification points computed by the pointer analysis-based approach can be thought of as "hints" to the user as to where provide specification.

In most cases, the user is likely to provide specification at program input points where he knows what the input strings may be. This is because at a reflective call it may be difficult to tell what all the constant class names that flow to it may be, as illustrated by Example 2. Generally, however, the user has a choice. For problematic reflective calls like those in `jedit` that produce a high number of specification points, a better strategy for the user may be to provide reflective specifications at the `Class.forName` *calls themselves* instead of laboriously going through all the specification points.

5.5 Casts for Reflection Resolution (Casts)

Type casts often provide a good first static approximation to what objects can be created at a given reflective creation site. There is a pretty significant increase in the number of `Class.forName` calls reported in Figure 6 in a few cases, including 93 newly discovered `Class.forName` calls in `gruntspud` that apprear due to a bigger call graph when reflective calls are resolved. In all cases, the majority of `Class.forName` calls have their targets at least partially resolved. In fact, as many as 95% of calls are resolved in the case of `jedit`.

As our experience with the Java reflection APIs would suggest, most `Class.newInstance` calls are post-dominated by a cast operation, often located within only a few lines of code of the `Class.newInstance` call. However, in our experiments, we have identified a number of `Class.newInstance` call sites, which were not dominated by a cast of any sort and therefore the return result of

Class.newInstance could not be constrained in any way. As it turns out, most of these unconstrained Class.newInstance call sites are located in the JDK and sun.∗ sources, Apache libraries, etc. Very few were found in application code.

The high number of unresolved calls in the JDK is due to the fact that reflection use in libraries tends to be highly generic and it is common to have "Class.newInstance wrappers" — methods that accept a class name as a string and return an object of that class, which is later cast to an appropriate type in the caller method. Since we rely on *intraprocedural* post-dominance, resolving these calls is beyond our scope. However, since such "wrapper" methods are typically called from multiple invocation sites and different sites can resolve to different types, it is unlikely that a precise approximation of the object type returned by Class.newInstance is possible in these cases at all.

Precision of Cast Information. Many reflective object creation sites are located in the JDK itself and are present in all applications we have analyzed. For example, method lookup in package sun.nio.cs.AbstractCharsetProvider reflectively creates a subclass of Charset and there are 53 different character sets defined in the system. In this case, the answer is precise because all of these charsets can conceivably be used depending on the application execution environment. In many cases, the cast approach is able to *uniquely* pinpoint the target of Class.newInstance calls based on cast information. For example, there is only one subclass of class sun.awt.shell.ShellFolderManager available to gruntspud, so, in order for the cast to succeed, it must be instantiated.

In general, however, the cast-based approach provides an imprecise upper bound on the call graph that needs to be analyzed. Because the results of Class.newInstance are occasionally cast to very wide types, such as java.lang.Cloneable, many potential subclasses can be instantiated at the Class.newInstance call site. The cast-based approach is likely to yield more precise results on applications that use Java generics, because those applications tend to use more narrow types when performing type casts.

5.6 Achieving a Sound Call Graph Approximation (Sound)

Providing a specification for unresolved reflective calls allows us to achieve a sound approximation of the call graph. In order to estimate the amount of effort required to come up with a specification for unresolved reflective calls, we decided to start with Points-to and add a reflection specification until the result became sound. Because providing a specification allows us to discover more of the call graph, two or three rounds of specification were required as new portions of the program became available. In practice, we would start without a specification and examine all unresolved calls and specification points corresponding to them. Then we would come up with a specification and feed it back to the call graph construction algorithm until the process converges.

Coming up with a specification is a difficult and error-prone task that requires looking at a large amount of source code. It took us about ten hours to incrementally devise an appropriate specification and ensure its completeness by rerunning the call graph construction algorithm. After providing a reflection specification stringing with Points-to, we then estimate how much of the user-provided specification can be avoided if we were to rely on type casts instead.

	Starting with Strings					Starting with Casts				
Benchmark	**Specs**	**Sites**	**Libs**	**App**	**Types/site**	**Specs**	**Sites**	**Libs**	**App**	**Types/site**
`jgap`	1,068	25	21	4	42.72	16	2	2	0	8.0
`freetts`	964	16	14	2	60.25	0	4	3	1	0.0
`gruntspud`	1,014	27	26	1	37.56	18	4	4	0	4.5
`jedit`	1,109	21	19	2	52.81	63	3	2	1	21.0
`columba`	1,006	22	21	1	45.73	16	2	2	0	8.0
`jfreechart`	1,342	21	21	0	63.90	18	4	4	0	4.5

Fig. 8. User-provided specification statistics

Specification Statistics. The first part of Figure 8 summarizes the effort needed to provide specifications to make the call graph sound. The second column shows the number of specifications of the form *reflective call site => type*, as exemplified by Figure 3. Columns 3–5 show the number of reflection calls sites covered by each specification, breaking them down into sites that located within library vs application code. As can be seen from the table, while the number of invocation sites for which specifications are necessary is always around 20, only a few are part of the application. Moreover, in the case of `jfreechart`, *all* of the calls requiring a specification are part of the library code.

Since almost all specification points are located in the JDK and library code, specification can be shared among different applications. Indeed, there are only 40 *unique* invocation sites requiring a specification across all the benchmarks. Column 6 shows the average number of types specified per reflective call site. Numbers in this columns are high because most reflective calls within the JDK can refer to a multitude of implementation classes.

The second part of Figure 8 estimates the specification effort required if were were to start with a cast-based call graph construction approach. As can be seen from columns 8–10, the number of `Class.forName` calls that are not constrained by a cast operation is quite small. There are, in fact, only 14 unique invocation sites — or about a third of invocation sites required for Points-to. This suggests that the the effort required to provide a specification to make Casts sound is considerably smaller than our original effort that starts with Points-to.

Specification Difficulties. In some cases, determining meaningful values to specify for `Class.forName` results is quite difficult, as shown by the example below.

Example 3. One of our benchmark applications, `jedit`, contains an embedded Bean shell, a Java source interpreter used to write editor macros. One of the calls to `Class.forName` within `jedit` takes parameters extracted from the Bean shell macros. In order to come up with a conservative superset of classes that may be invoked by the Bean shell interpreter for a given installation of `jedit`, we parse the scripts that are supplied with `jedit` to determine imported Java classes they have access to. (We should note that this specification is only sound for the default configuration of `jedit`; new classes may need to be added to the specification if new macros become available.) It took us a little under an hour to develop appropriate Perl scripts to do the parsing of 125 macros supplied with `jedit`. The `Class.forName` call can instantiate a total of 65 different types. □

We should emphasize that the conservativeness of the call graph depends on the conservativeness of the user-provided specification. If the specification missed potential relations, they will be also omitted from the call graph. Furthermore, a specification is typically only conservative for a given configuration of an application: if initialization files are different for a different program installation, the user-provided specification may no longer be conservative.

Remaining Unresolved Calls. Somewhat surprisingly, there are *still* some `Class.forName` calls that are not fully resolved given a user-provided specification, as can be seen from the last column in Figure 6. In fact, this is not a specification flaw: no valid specification is *possible* for those cases, as explained below.

Example 4. The audio API in the JDK includes method `javax.sound.sampled.AudioSystem.getDefaultServices`, which is not called in Java version 1.3 and above. A `Class.forName` call within that method resolves to constant `com.sun.media.sound.DefaultServices`, however, this class is absent in post-1.3 JDKs. However, since this method represents dead code, our answer is still sound. Similarly, other unresolved calls to `Class.forName` located within code that is not executed for the particular application configuration we are analyzing refer to classes specific to MacOS and unavailable on Linux, which is the platform we performed analysis on. In other cases, classes were unavailable for JDK version 1.4.2_08, which is the version we ran our analysis on. □

5.7 Effect of Reflection Resolution on Call Graph Size

Figure 9 compares the number of classes and methods across different analysis versions. Local analysis does not have any significant effect on the number of methods or classes in the call graph, even though most of the calls to `Class.forName` can be resolved with local analysis. This is due to the fact that

Classes						
Benchmark	None	Local	Points-to	Casts	Sound	
`jgap`	264	264	268	276	1,569	5.94
`freetts`	309	309	309	351	1,415	4.58
`gruntspud`	1,258	1,258	1,275	2,442	2,784	2.21
`jedit`	1,660	1,661	1,726	2,152	2,754	1.66
`columba`	961	962	966	1,151	2,339	2.43
`jfreechart`	884	881	886	1,560	2,340	2.65

Methods						
Benchmark	None	Local	Points-to	Casts	Sound	
`jgap`	1,013	1,014	1,038	1,075	6,676	6.58
`freetts`	1,357	1,358	1,358	1,545	5,499	4.05
`gruntspud`	7,321	7,321	7,448	14,164	14,368	1.96
`jedit`	11,230	11,231	11,523	13,487	16,003	1.43
`columba`	5,636	5,642	5,652	6,199	12,001	2.13
`jfreechart`	5,374	5,374	5,392	8,375	12,111	2.25

Fig. 9. Number of classes and methods in the call graph for different analysis versions

the vast majority of these calls are due to the use of the `T.class` idiom, which typically refer to classes that are already within the call graph. While these trivial calls are easy to resolve, it is the analysis of the other "hard" calls with a lot of potential targets that leads to a substantial increase in the call graph size.

Using Points-to increases the number of classes and methods in the call graph only moderately. The biggest increase in the number of methods occurs for `jedit` (293 methods). Using Casts leads to significantly bigger call graphs, especially for `gruntspud`, where the increase in the number of methods compared to None is almost two-fold.

The most noticeable increase in call graph size is observed in version Sound. Compared to None, the average increase in the number of classes is 3.2 times the original and the average increase for the number of methods is 3 times the original. The biggest increase in the number of methods occurs in `gruntspud`, with over 7,000 extra methods added to the graph.

Figure 9 also demonstrate that the lines of code metric is not always indicative of the size of the final call graph — programs are listed in the increasing order of line counts, yet, `jedit` and `gruntspud` are clearly the biggest benchmarks if we consider the method count. This can be attributed to the use of large libraries that ship with the application in binary form as well as considering a much larger portion of the JDK in version Sound compared to version None.

6 Related Work

General treatments of reflection in Java are given in Forman and Forman [1] and Guéhéneuc et al. [15]. The rest of the related work falls into the following broad categories: projects that explicitly deal with reflection in Java and other languages; approaches to call graph construction in Java; and finally, static and dynamic analysis algorithms that address the issue of dynamic class loading.

6.1 Reflection and Metadata Research

The metadata and reflection community has a long line of research originating in languages such as Scheme [16]. We only mention a few relevant projects here. The closest static analysis project to ours we are aware of is the work by Braux and Noyé on applying partial evaluation to reflection resolution for the purpose of optimization [17]. Their paper describes extensions to a standard partial evaluator to offer reflection support. The idea is to "compile away" reflective calls in Java programs, turning them into regular operations on objects and methods, given constraints on the concrete types of the object involved. The type constraints for performing specialization are provided by hand.

Our static analysis can be thought of as a tool for inferring such constraints, however, as our experimental results show, in many cases targets of reflective calls cannot be uniquely determined and so the benefits of specialization to optimize program execution may be limited. Braux and Noyé present a description of how their specialization approach may work on examples extracted from the JDK, but lacks a comprehensive experimental evaluation. In related work for languages other than Java, Ruf explores the use of partial evaluation as an optimization technique in the context of CLOS [18].

Specifying reflective targets is explicitly addressed in Jax [19]. Jax is concerned with reducing the size of Java applications in order to reduce download time; it reads in the class files that constitute a Java application, and performs a whole-program analysis to determine the components of the application that must be retained in order to preserve program behavior. Clearly, information about the true call graph is necessary to ensure that no relevant parts of the application are pruned away. Jax's approach to reflection is to employ user-provided specifications of reflective calls. To assist the user with writing complete specification files, Jax relies on dynamic instrumentation to discover the missing targets of reflective calls. Our analysis based on points-to information can be thought of as a tool for determining where to insert reflection specifications.

6.2 Call Graph Construction

A lot of effort has been spent of analyzing function pointers in C as well as virtual method calls in C++ and Java. We briefly mention some of the highlights of call graph construction algorithms for Java here. Grove et al. present a parameterized algorithmic framework for call graph construction [12,20]. They empirically assess a multitude of call graph construction algorithms by applying them to a suite of medium-sized programs written in Cecil and Java. Their experience with Java programs suggests that the effect of using context sensitivity for the task of call graph construction in Java yields only moderate improvements.

Tip and Palsberg propose a propagation-based algorithm for call graph construction and investigate the design space between existing algorithms for call graph construction such as 0-CFA and RTA, including RA, CHA, and four new ones [7]. Sundaresan et al. go beyond the tranditional RTA and CHA approaches in Java and and use type propagation for the purpose of obtaining a more precise call graph [21]. Their approach of using variable type analysis (VTA) is able to uniquely determine the targets of potentially polymorphic call sites in 32% to 94% of the cases. Agrawal et al. propose a demand-driven algorithm for call graph construction [22]. Their work is motivated by the need for just-in-time or dynamic compilation as well as program analysis used as part of software development environments. They demonstrate that their demand-driven technique has the same accuracy as the corresponding exhaustive technique. The reduction in the graph construction time depends upon the ratio of the cardinality of the set of influencing nodes to the set of all nodes.

6.3 Dynamic Analysis Approaches

Our work is motivated to a large extend by the need of error detection tool to have a static approximation of the true conservative call graph of the application. This largely precludes dynamic analysis that benefits optimizations such as method inlining and connectivity-based garbage collection.

A recent paper by Hirzel, Diwan, and Hind addresses the issues of dynamic class loading, native methods, and reflection in order to deal with the full complexity of Java in the implementation of a common pointer analysis [5]. Their approach involves converting the pointer analysis [6] into an online algorithm: they add constraints between analysis nodes as they are discovered at runtime.

Newly generated constraints cause re-computation and the results are propagated to analysis clients such as a method inliner and a garbage collector at runtime. Their approach leverages the class hierarchy analysis (CHA) to update the call graph. Our technique uses a more precise pointer analysis-based approach to call graph construction.

7 Conclusions

This paper presents the first static analysis for call graph construction in Java that addresses reflective calls. Our algorithm uses the results of a points-to analysis to determine potential reflective call targets. When the calls cannot be fully resolved, user-provided specification is requested. As an alternative to providing specification, type cast information can be used to provide a conservative approximation of reflective call targets.

We applied our static analysis techniques to the task of constructing call graphs for six large Java applications, some consisting of more than 190,000 lines of code. Our evaluation showed that as many as 95% of reflective `Class.forName` could at least partially be resolved to statically determined targets with the help of points-to results and cast information *without* providing any specification.

While most reflective calls are relatively easy to resolve statically, *precisely* interpreting some reflective calls requires a user-provided specification. Our pointer analysis-based approach also identified specification points — places in the program corresponding to file and system property read operations, etc., where user input is needed in order to obtain a full call graph. Our evaluation showed that the construction of a specification that makes the call graph conservative is a time-consuming and error-prone task. Fortunately, our cast-based approach can drastically reduce the specification burden placed on the user by providing a conservative, albeit potentially imprecise approximation of reflective targets.

Our experiments confirmed that ignoring reflection results in missing significant portions of the call graph, which is not something that effective static analysis tools can afford. While the local and points-to analysis techniques resulted in only a moderate increase in call graph size, using the cast-based approach resulted in call graphs with as many as 1.5 times more methods than the original call graph. Furthermore, providing a specification resulted in much larger conservative call graphs that were almost 7 times bigger than the original. For instance, in one our benchmark, an additional 7,047 methods were discovered in the conservative call graph version that were not present in the original.

References

1. Forman, I.R., Forman, N.: Java Reflection in Action. Manning Publications (2004)
2. Koved, L., Pistoia, M., Kershenbaum, A.: Access rights analysis for Java. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. (2002) 359 – 372
3. Reimer, D., Schonberg, E., Srinivas, K., Srinivasan, H., Alpern, B., Johnson, R.D., Kershenbaum, A., Koved, L.: SABER: Smart Analysis Based Error Reduction. In: Proceedings of International Symposium on Software Testing and Analysis. (2004) 243 – 251

4. Weimer, W., Necula, G.: Finding and preventing run-time error handling mistakes. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. (2004) 419 – 431
5. Hirzel, M., Diwan, A., Hind, M.: Pointer analysis in the presence of dynamic class loading. In: Proceedings of the European Conference on Object-Oriented Programming, Systems, Languages, and Applications. (2004) 96–122
6. Andersen, L.O.: Program analysis and specialization for the C programming language. PhD thesis, University of Copenhagen (1994)
7. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. ACM SIGPLAN Notices **35** (2000) 281–293
8. Livshits, B., Whaley, J., Lam, M.S.: Reflection analysis for Java, `http://suif.stanford.edu/~livshits/papers/tr/reflection_tr.pdf`. Technical report, Stanford University (2005)
9. Whaley, J., Lam, M.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proceedings of the ACM Conference on Programming Language Design and Implementation. (2004) 131 – 144
10. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. Lecture Notes in Computer Science **952** (1995) 77–101
11. Bacon, D.F.: Fast and Effective Optimization of Statically Typed Object-Oriented Languages. PhD thesis, University of California at Berkeley (1998)
12. Grove, D., Chambers, C.: A framework for call graph construction algorithms. ACM Trans. Program. Lang. Syst. **23** (2001) 685–746
13. Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: Proceedings of the ACM Symposium on Principles of Database Systems. (2005) 1 – 12
14. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
15. Guéhéneuc, Y.G., Cointe, P., Ségura-Devillechaise, M.: Java reflection exercises, correction, and FAQs. `http://www.yann-gael.gueheneuc.net/Work/Teaching/Documents/Practical-ReflectionCourse.doc.pdf` (2002)
16. Thiemann, P.: Towards partial evaluation of full Scheme. In: Reflection '96. (1996)
17. Braux, M., Noyé, J.: Towards partially evaluating reflection in Java. In: Proceedings of the ACM Workshop on Partial Evaluation and Semantics-based Program Manipulation. (1999) 2–11
18. Ruf, E.: Partial evaluation in reflective system implementations. In: Workshop on Reflection and Metalevel Architecture. (1993)
19. Tip, F., Laffra, C., Sweeney, P.F., Streeter, D.: Practical experience with an application extractor for Java. ACM SIGPLAN Notices **34** (1999) 292–305
20. Grove, D., DeFouw, G., Dean, J., Chambers, C.: Call graph construction in object-oriented languages. In: Proceedings of the ACM Conference on Object-oriented Programming, Systems, Languages, and Applications. (1997) 108–124
21. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for Java. ACM SIGPLAN Notices **35** (2000) 264–280
22. Agrawal, G., Li, J., Su, Q.: Evaluating a demand driven technique for call graph construction. In: Computational Complexity. (2002) 29–45

Lightweight Family Polymorphism

Atsushi Igarashi[1], Chieri Saito[1], and Mirko Viroli[2]

[1] Kyoto University, Japan
{igarashi, saito}@kuis.kyoto-u.ac.jp
[2] Alma Mater Studiorum – Università di Bologna a Cesena, Italy
mviroli@deis.unibo.it

Abstract. Family polymorphism has been proposed for object-oriented languages as a solution to supporting reusable yet type-safe mutually recursive classes. A key idea of family polymorphism is the notion of families, which are used to group mutually recursive classes. In the original proposal, due to the design decision that families are represented by objects, dependent types had to be introduced, resulting in a rather complex type system. In this paper, we propose a simpler solution of *lightweight* family polymorphism, based on the idea that families are represented by classes rather than objects. This change makes the type system significantly simpler without losing much expressibility of the language. Moreover, "family-polymorphic" methods now take a form of parametric methods; thus it is easy to apply the Java-style type inference. To rigorously show that our approach is safe, we formalize the set of language features on top of Featherweight Java and prove the type system is sound. An algorithm of type inference for family-polymorphic method invocations is also formalized and proved to be correct.

1 Introduction

Mismatch between Mutually Recursive Classes and Simple Inheritance. It is fairly well-known that, in object-oriented languages with simple name-based type systems such as C++ or Java, mutually recursive class definitions and extension by inheritance do not fit very well. Since classes are usually closed entities in a program, mutually recursive classes here really mean a set of classes whose method *signatures* refer to each other by their *names*. Thus, different sets of mutually recursive classes necessarily have different signatures, even though their structures are similar. On the other hand, in C++ or Java, it is not allowed to inherit a method from the superclass with a different signature (in fact, it is not safe in general to allow covariant change of method parameter types). As a result, deriving subclasses of mutually recursive classes yields another set of classes that do *not* refer to each other and, worse, this mismatch is often resolved by typecasting, which is a potentially unsafe operation (not to say unsafe, an exception may be raised). A lot of studies [6,8,11,15,17,20,3,19] have been recently done to develop a language mechanism with a static type system that allows "right" extension of mutually recursive classes without resorting to typecasting or other unsafe features.

Family Polymorphism. Erik Ernst [11] has recently coined the term "family polymorphism" for a particular programming style using virtual classes [16] of `gbeta` [10] and applied it to solve the above-mentioned problem of mutually recursive classes.

In his proposal, mutually recursive classes are programmed as nested class members of another (top-level) class. Those member classes are virtual in the same sense as virtual methods—a reference to a class member is resolved at run-time. Thus, the meaning of mutual references to class names will change when a subclass of the enclosing class is derived and those member classes are inherited. This late-binding of class names makes it possible to reuse implementation without the mismatch described above. The term "family" refers to such a set of mutually recursive classes grouped inside another class. He has also shown how a method that can uniformly work for different families can be written in a safe way: such "family-polymorphic" methods take as arguments not only instances of mutually recursive classes but also the identity of the family that they belong to, so that semantical analysis (or a static type checker) can check if those instances really belong to the same family.

Although family polymorphism seems very powerful, we feel that there may be a simpler solution to the present problem. In particular, in `gbeta`, nested classes really are members (or, more precisely, attributes) of an *object*, so types for mutually recursive classes include as part object references, which serve as identifiers of families. As a result, the semantical analysis of `gbeta` essentially involves a dependent type system [1,19], which is rather complex (especially in the presence of side effects).

Contributions of the Paper. We identify a minimal, *lightweight* set of language features to solve the problem of typing mutually recursive classes, rather than introduce a new advanced mechanism. As done in elsewhere [15], we adopt what we call the "classes-as-families" principle, in which families are identified with classes, which are static entities, rather than objects, which are dynamic. Although it loses some expressibility, a similar style of programming is still possible. Moreover, we take the approach that inheritance is not subtyping, for type safety reasons, and also avoid exact types [6], which are often deemed important in this context. These decisions simplify the type system a lot, making much easier a type soundness argument and application to practical languages such as Java. As a byproduct, we can view family polymorphic methods as a kind of parametric methods found e.g. in Java generics and find that the technique of type argument synthesis as in GJ and Java 5.0 [2,18] can be extended to our proposal as well.

Other technical contributions of the present paper can be summarized as follows:

- Simplification of the type system for family polymorphism with the support for family-polymorphic methods;
- A rigorous discussion of the safety issues by the development of a formal model called .FJ (read "dot FJ") of lightweight polymorphism, on top of

Featherweight Java [14] by Igarashi, Pierce, and Wadler, with a correctness theorem of the type system; and

– An algorithm of type argument synthesis for family-polymorphic methods and its correctness theorem.

The Rest of This Paper. After Section 2 presents the overview of our language constructs through the standard example of graphs, in Section 3, we formalize those mechanisms as the calculus .FJ and discuss its type safety. Then, Section 4 presents a type inference algorithm for family-polymorphic method invocations and discuss its correctness. Section 5 discusses related work, and Section 6 concludes. For brevity, we omit proofs of theorems, which appear in a full version available at `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/`.

2 Programming Lightweight Family Polymorphism

We start by informally describing the main aspects of the language constructs we study in this paper, used to support lightweight family polymorphism. To this end, we consider as a reference the example in [11], properly adapted to fit our "classes-as-families" principle.

This example features a *family* (or group) `Graph`, containing the classes `Node` and `Edge`, which are the *members* of the family, and are used as components to build graph instances. As typically happens, members of the same family can mutually refer to each other: in our example for instance, each node holds a reference to connected edges, while each edge holds a reference to its source and destination nodes. Now suppose we are interested in defining a new family `ColorWeightGraph`, used to define graphs with colored nodes and weighted edges—nodes and edges with the new fields called `color` and `weight`, respectively—with the property that the weight of an edge depends on the color of its source and destination nodes. Note that in this way the the members of the family `Graph` are not compatible with those of family `ColorWeightGraph` in the sense that an edge of a `ColorWeightGraph` cannot be used in a plain `Graph`. Nevertheless, to achieve code reuse, we would like to define the family `ColorWeightGraph` as an extension of the family `Graph`, and declare a member `Node` which automatically inherits all the attributes (fields and methods) of `Node` in `Graph`, and similarly for member `Edge`. Moreover, as advocated by the family polymorphism idea, we would like classes `Node` and `Edge` in `ColorWeightGraph` to mutually refer to each other automatically, as opposed to those solutions exploiting simple inheritance where class `Node` of `ColorWeightGraph` would refer to `Edge` of `Graph`—thus requiring an extensive use of typecasts.

2.1 Nested Classes, Relative Path Types, and Extension of Families

This graph example can be programmed using our lightweight family polymorphism solution as reported in Figure 1, whose code adheres to a Java-like syntax—which is also the basis for the syntax of the calculus .FJ we introduce in Section 3.

```
class Graph {
    static class Node {
        .Edge[] es=new .Edge[10]; int i=0;
        void add(.Edge e) { es[i++] = e; }}
    static class Edge {
        .Node src, dst;
        void connect(.Node s, .Node d) {
            src = s; dst = d; s.add(this); d.add(this);
}   }   }
class ColorWeightGraph extends Graph {
    static class Node { Color color; }
    static class Edge {
        int weight;
        void connect(.Node s, .Node d) {
            weight = f(s.color, d.color); super.connect(s, d);
}   }   }
```

```
Graph.Edge e; Graph.Node n1, n2;
ColorWeightGraph.Edge we; ColorWeightGraph.Node cn1, cn2;
e.connect(n1, n2);     // 1: OK
we.connect(cn1, cn2); // 2: OK
we.connect(n1, cn2);  // 3: compile-time error
e.connect(n1, cn2);   // 4: compile-time error
```

```
<G extends Graph>
  void connectAll(G.Edge[] es, G.Node n1, G.Node n2){
    for (int i: es) es[i].connect(n1,n2); }

Graph.Edge[] ges;             Graph.Node gn1,gn2;
ColorWeightGraph.Edge[] ces; ColorWeightGraph.Node cn1,cn2;
connectAll(ges, gn1, gn2);   // G as Graph
connectAll(ces, cn1, cn2);   // G as ColorWeightGraph
connectAll(ces, gn1, gn2);   // compile-time error
```

Fig. 1. `Graph` and `ColorWeightGraph` Classes

The first idea is to represent families as (top-level) classes, and their members as nested classes. Note that in particular we relied on the syntax of Java `static` member classes, which provide a grouping mechanism suitable to define a family. In spite of this similarity, however, we shall give a different semantics to those member classes, in order to support family polymorphism. The types of nodes and edges of class (family) `Graph` are denoted by notations `Graph.Node` and `Graph.Edge` which we call *absolute path types*. Whereas such types are useful outside the family to declare variables and to create instances of such member classes, we do not use them to specify mutual references of family members. The notations `.Node` and `.Graph` are instead introduced for this purpose, meaning "member `Node` in the current family" and "member `Edge` in the current family," respectively. We call such types *relative path types*: this terminology is justified by noting that while the notation for a type $\mathtt{C}_1\mathtt{.C}_2$ resembles an absolute directory path $\mathtt{/d}_1\mathtt{/d}_2$, notation $\mathtt{.C}_2$ resembles the relative directory path $\mathtt{../d}_2$.

The importance of relative path types becomes clear when introducing the concept of family extension. To define the new family `ColorWeightGraph`, a new class `ColorWeightGraph` is declared to extend `Graph` and providing the member classes `Node` and `Edge`. Such new members, identified outside their family by the absolute path types `ColorWeightGraph.Node` and `ColorWeightGraph.Edge`, will inherit all the attributes of classes `Graph.Node` and `Graph.Edge`, respectively. In particular, `ColorWeightGraph.Edge` will inherit method `connect()` from `Graph.Edge`, and can therefore override it as shown in the reference code, and even redirect calls by the invocation `super.connect()`. However, since `connect()` is declared to accept two arguments of relative path type `.Node`, it will accept a `Graph.Node` when invoked on a `Graph.Edge`, and a `ColorWeightGraph.Node` when invoked on a `ColorWeightGraph.Edge`. Notice that relative path types are essential to realize family polymorphism, as they guarantee members of the extended family to mutually refer to each other, and not to refer to a different (super) family.

2.2 Inheritance Is Not Subtyping for Member Classes

This covariance schema for relative path types—they change as we move from a family to a subfamily—resembles and extends the construct of *ThisType* [5], used to make method signatures of self-referencing classes change covariantly through inheritance hierarchies. As well known, however, such a covariance schema prevents inheritance and substitutability from correctly working together as happens in most of common object-oriented languages. In particular, when a relative path type is used as an argument type to a method in a family member, as in method `connect()` of class `Edge`, they prevent its instances from being substituted for those in the superfamily, even though the proper inheritance relation is supported. The following code fragment reveals this problem:

```
// If ColorWeightGraph.Edge were substitutable for Graph.Edge
Graph.Edge e=new ColorWeightGraph.Edge();
Graph.Node n1=new Graph.Node();
Graph.Node n2=new Graph.Node();
e.connect(n1,n2);  // Unsafe!!
```

If class `ColorWeightGraph.Edge` could be substituted for `Graph.Edge`, then it could happen to invoke `connect()` on a `ColorWeightGraph.Edge` passing some `Graph.Node` as elements. Such an invocation would lead to the attempt of accessing field `color` on an object of class `Graph.Node`, which does *not* have such a field!

To prevent this form of unsoundness, our lightweight family polymorphism solution disallows such substitutability by adopting an "inheritance without subtyping" approach for family members. Applied to our graph example, it means that while `ColorWeightGraph.Node` inherits all the attributes of `Graph.Node` (for `ColorWeightGraph` extends `Graph`), `ColorWeightGraph.Node` is *not* a subtype of `Graph.Node`. As a result of this choice, we can correctly typecheck the invocation of methods in member classes. In the client code in the middle of Figure 1, the first two invocations are correct as node arguments belong to the same family of the receiver edge, but the third and fourth are (statically) rejected, as we are passing as argument a node belonging to a family different from the receiver edge: in other words, `Graph.Node` and `ColorWeightGraph.Node` are not in the subtype relation.

2.3 Family-Polymorphic Methods as Parametric Methods

To fully exploit the benefits of family polymorphism it should be possible to write so-called *family-polymorphic methods*—methods that can work uniformly over different families. As an example, we consider the method `connectAll()` that takes as input an array of edges and two nodes of *any* family and connects each edge to the two nodes. In our language this is realized through parametric methods as shown at the bottom of Figure 1. Method `connectAll()` is defined as parametric in a type `G`—which represents the family used for each invocation—with upper-bound `Graph` and correspondingly the arguments are of type `G.Edge[]`, `G.Node` and `G.Node` respectively. As a result, in the first invocation of the example code, by passing edges and nodes of family `Graph` the compiler would infer the type `Graph` for `G`, and similarly in the second invocation infers `ColorWeightGraph`. Finally, in the third invocation no type can be inferred for `G`, since for no `G`, types `G.Edge` and `G.Node` match `ColorWeightGraph.Edge` and `Graph.Node`, respectively.

It may be worth noting that we do not allow relative path types to appear directly in a top-level class: for instance, `.Node` cannot appear in `Graph` or `ColorWeightGraph`. This is because allowing it would prevent us from assuming that `ColorWeightGraph` is a subtype of `Graph` (for much the same reason as above), which is used to realize family-polymorphic methods.

3 .FJ: A Formal Model of Lightweight Family Polymorphism

In this section, we formalize the ideas described in the previous section, namely, nested classes with simultaneous extension, relative path types and family-polymorphic methods as a small calculus named .FJ based on Featherweight Java [14], a functional core of class-based object-oriented languages. After formally defining the syntax (Section 3.1), type system (Sections 3.2 and 3.3), and operational semantics (Section 3.4) of .FJ, we show a type soundness result (Section 3.5).

For simplicity, we deal with only a single level of nesting, as opposed to Java, which allows arbitrary levels of nesting. We believe that, for programming family polymorphism, little expressiveness is lost by this restriction, though a language with arbitrarily deep nesting would be interesting. Although they are easy to add, typecasts—which appear in Featherweight Java and are essential to discuss erasure compilation of generics—are dropped since one of our aims here is to show the programming as in the previous section is possible without typecasts. In .FJ, every parametric method invocation has to provide its type arguments—type inference will be discussed in Section 4. Method invocation on `super` is also omitted since directly formalizing `super` would require several global changes to the calculus, due to the fact that `super` invocation is not virtual [13] and, more importantly, it does not really pose a new typing challenge. Invocations on `super` work for much the same reason as invocations of inherited methods on `this` work.

3.1 Syntax

The abstract syntax of top-level/nested class declarations, constructor declarations, method declarations, and expressions of .FJ is given in Figure 2. Here, the metavariables `C`, `D`, and `E` range over (simple) class names; `X` and `Y` range over type variable names; `f` and `g` range over field names; `m` ranges over method names; `x` ranges over variables.

We put an over-line for a possibly empty sequence. Furthermore, we abbreviate pairs of sequences in a similar way, writing "$\overline{\mathtt{T}}\ \overline{\mathtt{f}}$" for "$\mathtt{T}_1\ \mathtt{f}_1,\ldots,\mathtt{T}_n\ \mathtt{f}_n$", where n is the length of $\overline{\mathtt{T}}$ and $\overline{\mathtt{f}}$, and "$\mathtt{this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\mathtt{;}$" as shorthand for "$\mathtt{this.f}_1\mathtt{=f}_1\mathtt{;}\ldots\mathtt{;this.f}_n\mathtt{=f}_n\mathtt{;}$" and so on. Sequences of type variables, field declarations, parameter names, and method declarations are assumed to contain no duplicate names. We write the empty sequence as • and denote concatenation of sequences using a comma.

A family name `P`, used as a type argument to family-polymorphic methods, is either a top-level class name or a type variable. Absolute class names can be used to instantiate objects, so they play the role of run-time types of objects. A type can be an absolute path type `P` or `P.C`, or a relative path type `.C`. A top-level class declaration consists of its name, its superclass, field declarations, a constructor, methods, and nested classes. The symbol $\triangleleft$ is read `extends`. On the other hand, a nested class does not have an `extends` clause since the class from which it inherits is implicitly determined. We have dropped the keyword `static`, used in the previous section, for conciseness. As in Featherweight Java, a constructor is given in a stylized syntax and just takes initial (and final) values for the fields and assigns them to corresponding fields. A method declaration can be parameterized by type variables, whose bounds are top-level class (i.e., family) names. Since the language is functional, the body of a method is a single `return` statement. An expression is either a variable, field access, method invocation, or object creation. We assume that the set of variables includes the special variable `this`, which cannot be used as the name of a parameter to a method.

A class table *CT* is a mapping from absolute class names `A` to (top-level or nested) class declarations. A program is a pair (*CT*, `e`) of a class table and an expression. To lighten the notation in what follows, we always assume a

$\mathtt{P,Q}$	$::= \mathtt{C} \mid \mathtt{X}$	family names
$\mathtt{A,B}$	$::= \mathtt{C} \mid \mathtt{C.C}$	absolute class names
$\mathtt{S,T,U}$	$::= \mathtt{P} \mid \mathtt{P.C} \mid \mathtt{.C}$	types
$\mathtt{L}$	$::= \mathtt{class\ C} \triangleleft \mathtt{C}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\ \overline{\mathtt{N}}\}$	top class declarations
$\mathtt{K}$	$::= \mathtt{C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{f}});\ \mathtt{this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\}$	constructor declarations
$\mathtt{M}$	$::= \mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}\mathtt{>T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}$	method declarations
$\mathtt{N}$	$::= \mathtt{class\ C}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}$	nested class declarations
$\mathtt{d,e}$	$::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.<}\overline{\mathtt{P}}\mathtt{>m}(\overline{\mathtt{e}}) \mid \mathtt{new\ A}(\overline{\mathtt{e}})$	expressions
$\mathtt{v}$	$::= \mathtt{new\ A}(\overline{\mathtt{v}})$	values

Fig. 2. .FJ: Syntax

fixed class table *CT*. As in Featherweight Java, we assume that `Object` has no members and its definition does *not* appear in the class table.

3.2 Lookup Functions

Before proceeding to the type system, we give functions to look up field or method definitions. The function *fields*(`A`) returns a sequence $\overline{\mathtt{T}}\ \overline{\mathtt{f}}$ of field names of the class `A` with their types. The function *mtype*(`m`, `A`) takes a method name and a class name as input and returns the corresponding method signature of the form $\mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T}_0$, in which $\overline{\mathtt{X}}$ are bound. They are defined by the rules in Figure 3. Here, $\mathtt{m} \notin \overline{\mathtt{M}}$ (and $\mathtt{E} \notin \overline{\mathtt{N}}$) means the method of name `m` (and the nested class of name `E`, respectively) does not exist in $\overline{\mathtt{M}}$ (and $\overline{\mathtt{N}}$, respectively).

As mentioned before, `Object` does not have any fields, methods, or nested classes, so *fields*(`Object`) = *fields*(`Object.C`) = • for any `C`, and *mtype*(`m`, `Object`) and *mtype*(`m`, `Object.C`) are undefined. The definitions are straightforward extensions of the ones in Featherweight Java. Interesting rules are the last rules: when a nested class `C.E` does not exist, lookup proceeds in the nested class of the same name `E` in the superclass of the enclosing class `C`. Notice that, when the method definition is found in a superclass, relative path types, whose meaning depends on the type of the receiver, in the method signature remain unchanged; they are resolved in typing rules. Also note that, by this definition, *fields*(`C.D`)

Field Lookup:

$$\mathit{fields}(\mathtt{Object}) = \bullet$$

$$\frac{\mathtt{class\ C} \triangleleft \mathtt{D\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;..\}} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}, \overline{\mathtt{T}}\ \overline{\mathtt{f}}}$$

$$\mathit{fields}(\mathtt{Object.C}) = \bullet$$

$$\frac{\mathtt{class\ C} \triangleleft \mathtt{D\{..\ class\ E\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;..\}..\}} \qquad \mathit{fields}(\mathtt{D.E}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C.E}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}, \overline{\mathtt{T}}\ \overline{\mathtt{f}}}$$

$$\frac{\mathtt{class\ C} \triangleleft \mathtt{D\ \{..}\overline{\mathtt{N}}\mathtt{\}} \quad \mathtt{E} \notin \overline{\mathtt{N}} \qquad \mathit{fields}(\mathtt{D.E}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C.E}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}$$

Method Type Lookup:

$$\frac{\mathtt{class\ C} \triangleleft \mathtt{D\ \{..}\overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>T}_0\ \mathtt{m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}} \in \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T}_0}$$

$$\frac{\mathtt{class\ C} \triangleleft \mathtt{D\ \{..}\overline{\mathtt{M}}\mathtt{..\}} \quad \mathtt{m} \notin \overline{\mathtt{M}} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{D}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T}_0}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T}_0}$$

$$\frac{\mathtt{class\ C} \triangleleft \mathtt{D\ \{..\ class\ E\ \{..}\overline{\mathtt{M}}\mathtt{\}..\}} \qquad \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>T}_0\ \mathtt{m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}} \in \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C.E}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T}_0}$$

$$\frac{\mathtt{class\ C} \triangleleft \mathtt{D\ \{..\ class\ E\ \{..}\overline{\mathtt{M}}\mathtt{\}..\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \quad \mathit{mtype}(\mathtt{m}, \mathtt{D.E}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T}_0}{\mathit{mtype}(\mathtt{m}, \mathtt{C.E}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T}_0}$$

$$\frac{\mathtt{class\ C} \triangleleft \mathtt{D\ \{..}\overline{\mathtt{N}}\mathtt{\}} \quad \mathtt{E} \notin \overline{\mathtt{N}} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{D.E}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T}_0}{\mathit{mtype}(\mathtt{m}, \mathtt{C.E}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T}_0}$$

Fig. 3. .FJ: Lookup functions

is defined for any D if $\mathtt{C} \in \mathit{dom}(CT)$. It does no harm since we never ask fields of such non-existing classes when all types in the class table are well formed. We also think it would clutter the presentation to give a definition in which *fields*(C.D) is undefined when C or its superclasses do not have D.

3.3 Type System

The main judgments of the type system consist of one for subtyping $\Delta \vdash \mathtt{S} <: \mathtt{T}$, one for type well-formedness $\Delta \vdash_{\mathtt{A}} \mathtt{T}\ \mathrm{ok}$, and one for typing $\Delta; \Gamma \vdash_{\mathtt{A}} \mathtt{e} : \mathtt{T}$. Here, Δ is a *bound environment*, which is a finite mapping from type variables to their bounds, written $\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{C}}$; Γ is a *type environment*, which is a finite mapping from variables to types, written $\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}}$. Since we are not concerned with more general forms of bounded polymorphism, upper bounds are always top-level class names. By slight abuse of notation, we write $\Delta(\mathtt{T})$ for the upper bound of T in Δ, defined by: $\Delta(\mathtt{A}) = \mathtt{A}$ and $\Delta(\mathtt{X.C}) = \Delta(\mathtt{X})\mathtt{.C}$. We never ask the upper bound of a relative path type, so $\Delta(\mathtt{.C})$ is undefined. We abbreviate a sequence of judgments in the obvious way: $\Delta \vdash \mathtt{S}_1 <: \mathtt{T}_1, \ldots, \Delta \vdash \mathtt{S}_n <: \mathtt{T}_n$ to $\Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}$; $\Delta \vdash_{\mathtt{A}} \mathtt{T}_1\ \mathrm{ok}, \ldots,$ $\Delta \vdash_{\mathtt{A}} \mathtt{T}_n\ \mathrm{ok}$ to $\Delta \vdash_{\mathtt{A}} \overline{\mathtt{T}}\ \mathrm{ok}$; and $\Delta; \Gamma \vdash_{\mathtt{A}} \mathtt{e}_1 : \mathtt{T}_1, \ldots, \Delta; \Gamma \vdash_{\mathtt{A}} \mathtt{e}_n : \mathtt{T}_n$ to $\Delta; \Gamma \vdash_{\mathtt{A}} \overline{\mathtt{e}} : \overline{\mathtt{T}}$.

Subtyping. The subtyping judgment $\Delta \vdash \mathtt{S} <: \mathtt{T}$, read as "S is subtype of T under Δ," is defined in Figure 4. This relation is the reflexive and transitive closure of the `extends` relation with `Object` being the top type. Note that a nested class, which does not have the `extends` clause, has only a trivial super/subtype, which is itself, even if some members are inherited from another (nested) class.

Type Well-formedness. The type well-formedness judgment $\Delta \vdash_{\mathtt{A}} \mathtt{T}\ \mathrm{ok}$, read as "T is a well formed type in (the body of) class A under Δ." The rules for well-formed types appear also in Figure 4. `Object` and class names in the domain of the class table are well formed. Moreover, a nested class name C.E is well formed if E is inherited from C's superclass D. Type X (possibly with a suffix) is well formed if its upper bound (with the suffix) is well formed. Finally, a relative path type .E is well formed in a nested class C.D if C.E is well formed.

Typing. Typing requires another auxiliary (but important) definition. The *resolution* T@S of T at S, which intuitively denotes the class name that T refers to in a given class S, is defined by:

Subtyping:

$$\Delta \vdash \mathtt{T} <: \mathtt{T} \qquad \Delta \vdash \mathtt{X} <: \Delta(\mathtt{X})$$

$$\Delta \vdash \mathtt{T} <: \mathtt{Object} \qquad \frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{..\}}{\Delta \vdash \mathtt{C} <: \mathtt{D}}$$

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \qquad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}}$$

Type Well-formedness:

$$\Delta \vdash_{\mathtt{A}} \mathtt{Object}\ \mathrm{ok} \qquad \frac{\Delta(\mathtt{P}) \in \mathit{dom}(CT)}{\Delta \vdash_{\mathtt{A}} \mathtt{P}\ \mathrm{ok}}$$

$$\frac{\begin{array}{c}\mathtt{class\ C} \lhd \mathtt{D}\{..\ \overline{\mathtt{N}}\} \\ \mathtt{E} \notin \overline{\mathtt{N}} \qquad \Delta \vdash_{\mathtt{A}} \mathtt{D.E}\ \mathrm{ok}\end{array}}{\Delta \vdash_{\mathtt{A}} \mathtt{C.E}\ \mathrm{ok}} \qquad \frac{\Delta \vdash_{\mathtt{C.D}} \mathtt{C.E}\ \mathrm{ok}}{\Delta \vdash_{\mathtt{C.D}} \mathtt{.E}\ \mathrm{ok}}$$

Fig. 4. .FJ: Subtyping and type well-formedness

Expression Typing:

$$\Delta;\Gamma \vdash_{\mathtt{A}} \mathtt{x}:\Gamma(\mathtt{x}) \qquad \text{(T-Var)}$$

$$\frac{\Delta;\Gamma \vdash_{\mathtt{A}} \mathtt{e}_0:\mathtt{T}_0 \qquad \mathit{fields}(\Delta(\mathtt{T}_0@\mathtt{A})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Delta;\Gamma \vdash_{\mathtt{A}} \mathtt{e}_0.\mathtt{f}_i:\mathtt{T}_i@\mathtt{T}_0} \qquad \text{(T-Field)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash_{\mathtt{A}} \mathtt{e}_0 : \mathtt{T}_0 \qquad \mathit{mtype}(\mathtt{m}, \Delta(\mathtt{T}_0@\mathtt{A})) = \mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0 \\ \Delta \vdash \overline{\mathtt{P}}\ \mathtt{<:}\ \overline{\mathtt{C}} \qquad \Delta;\Gamma \vdash_{\mathtt{A}} \overline{\mathtt{e}}:\overline{\mathtt{T}} \qquad \Delta \vdash \overline{\mathtt{T}}\ \mathtt{<:}([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{U}})@\mathtt{T}_0\end{array}}{\Delta;\Gamma \vdash_{\mathtt{A}} \mathtt{e}_0.\mathtt{<}\overline{\mathtt{P}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} : ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{U}_0)@\mathtt{T}_0} \qquad \text{(T-Invk)}$$

$$\frac{\mathit{fields}(\mathtt{A}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Delta;\Gamma \vdash_{\mathtt{A}} \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \Delta \vdash \overline{\mathtt{U}}\ \mathtt{<:}\ (\overline{\mathtt{T}}@\mathtt{A}_0)}{\Delta;\Gamma \vdash_{\mathtt{A}} \mathtt{new}\ \mathtt{A}_0\mathtt{(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{A}_0} \qquad \text{(T-New)}$$

Method Typing:

$$\frac{\begin{array}{c}\Delta = \overline{\mathtt{X}}\ \mathtt{<:}\ \overline{\mathtt{C}} \qquad \Delta;\overline{\mathtt{x}}:\overline{\mathtt{T}}, \mathtt{this}:\mathit{thisty}(\mathtt{A}) \vdash_{\mathtt{A}} \mathtt{e}_0:\mathtt{U}_0 \\ \Delta \vdash \mathtt{U}_0\ \mathtt{<:}\ \mathtt{T}_0 \qquad \Delta \vdash_{\mathtt{A}} \mathtt{T}_0, \overline{\mathtt{T}}, \overline{\mathtt{C}}\ \text{ok} \\ \text{if } \mathit{mtype}(\mathtt{m}, \mathit{supcls}(\mathtt{A})) = \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{D}}\mathtt{>}\overline{\mathtt{S}}\rightarrow\mathtt{S}_0, \text{ then } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \overline{\mathtt{T}},\mathtt{T}_0 = [\overline{\mathtt{X}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}},\mathtt{S}_0)\end{array}}{\vdash_{\mathtt{A}} \mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{C}}\mathtt{>}\mathtt{T}_0\ \mathtt{m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{return}\ \mathtt{e}_0\mathtt{;\}}\ \text{ok}} \qquad \text{(T-Method)}$$

Class Typing:

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{E(}\overline{\mathtt{U}}\ \overline{\mathtt{g}}\mathtt{,}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\{super(}\overline{\mathtt{g}}\mathtt{);this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\mathtt{;\}} \\ \mathit{fields}(\mathit{supcls}(\mathtt{C.E})) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \vdash_{\mathtt{C.E}} \overline{\mathtt{M}}\ \text{ok} \qquad \emptyset \vdash_{\mathtt{C.E}} \overline{\mathtt{T}}\ \text{ok}\end{array}}{\vdash_{\mathtt{C}} \mathtt{class}\ \mathtt{E\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \mathtt{K}\ \overline{\mathtt{M}}\mathtt{\}}\ \text{ok}} \qquad \text{(T-NClass)}$$

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{C(}\overline{\mathtt{U}}\ \overline{\mathtt{g}}\mathtt{,}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\{super(}\overline{\mathtt{g}}\mathtt{);this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\mathtt{;\}} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \vdash_{\mathtt{C}} \overline{\mathtt{M}}\ \text{ok} \qquad \vdash_{\mathtt{C}} \overline{\mathtt{N}}\ \text{ok} \qquad \emptyset \vdash_{\mathtt{C}} \overline{\mathtt{T}}, \mathtt{D}\ \text{ok}\end{array}}{\vdash \mathtt{class}\ \mathtt{C} \lhd \mathtt{D\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \mathtt{K}\ \overline{\mathtt{M}}\ \overline{\mathtt{N}}\mathtt{\}}\ \text{ok}} \qquad \text{(T-TClass)}$$

Fig. 5. .FJ: Typing

$$\mathtt{.D}@\mathtt{P.C} = \mathtt{P.D} \qquad \mathtt{.D}@\mathtt{.C} = \mathtt{.D} \qquad \mathtt{P}@\mathtt{T} = \mathtt{P} \qquad \mathtt{P.C}@\mathtt{T} = \mathtt{P.C}.$$

The only interesting case is the first clause: It means that a relative path type `.D` in `P.C` refers to `P.D`—it resembles the command `cd` of UNIX shells: `cd ../D` changes the current directory from `P/C` to `P/D`. For example, `.Edge@Graph.Node` = `Graph.Edge`. Note that `.D@C` and `.D@X` are undefined since a relative path type is not allowed to appear in top-level classes.

The typing judgment for expressions is of the form $\Delta;\Gamma \vdash_{\mathtt{A}} \mathtt{e}:\mathtt{T}$, read as "under bound environment Δ and type environment Γ, expression `e` has type `T` in class `A`." Typing rules are given in Figure 5. Interesting rules are T-Field and T-Invk, although the basic idea is as usual—for example, in T-Field, the field types are retrieved from the receiver's type $\mathtt{T}_0$, and the corresponding type of the accessed field is the type of the whole expression. We need some tricks to deal with relative path types (and type variables): if the receiver's type $\mathtt{T}_0$ is a relative path type, it has to be resolved in $\mathtt{A}$, the class in which $\mathtt{e}$ appears; a type variable is taken to its upper bound by $\Delta(\cdot)$. Moreover, if the field type is a relative path type, it is resolved in the *receiver's type*. For example, if $\mathit{fields}(\mathtt{CWGraph.Node}) = \mathtt{.Edge\ edg}$ and $\Gamma = \mathtt{x{:}CWGraph.Node,\ y{:}.Node}$, then

$$\Delta;\Gamma \vdash_{\mathtt{CWGraph.Node}} \mathtt{x.edg\ :\ CWGraph.Edge} \text{ and}$$
$$\Delta;\Gamma \vdash_{\mathtt{CWGraph.Node}} \mathtt{y.edg\ :\ .Edge}.$$

In this way, accessing a field of relative path type gives a relative path type only when the receiver is also given a relative path type. Similarly, in T-Invk, the method type is retrieved from the receiver's type; then, it is checked whether the given type arguments are subtypes of bounds $\overline{\mathtt{C}}$ of formal type parameters and the types of actual value arguments are subtypes of those of formal parameters, where type arguments are substituted for variables. For example, if $\mathit{mtype}(\mathtt{connectAll}, \mathtt{C}) = \mathtt{<G{\triangleleft}Graph>(G.Node,G.Edge)}{\rightarrow}\mathtt{void}$, then

$$\Delta; \mathtt{x{:}C}, \mathtt{n{:}CWGraph.Node}, \mathtt{e{:}CWGraph.Edge} \vdash_{\mathtt{A}}$$
$$\mathtt{x.connectAll{<}CWGraph{>}(n,e)} : \mathtt{void}.$$

A judgment for method typing is written $\vdash_{\mathtt{A}} \mathtt{M}$ ok, and derived by T-Method. Here, $\mathit{thisty}(\mathtt{A})$ and $\mathit{supcls}(\mathtt{A})$ are defined by:

$$\mathit{thisty}(\mathtt{C}) = \mathtt{C} \qquad \mathit{supcls}(\mathtt{C}) = \mathtt{D}$$
$$\mathit{thisty}(\mathtt{C.E}) = \mathtt{.E} \qquad \mathit{supcls}(\mathtt{C.E}) = \mathtt{D.E}$$

where $\mathtt{class\ C} \triangleleft \mathtt{D\{..\}}$. It is checked that the body of the method is well typed under the bound and type environments obtained from the definition. Note that `this` of a nested class is given a relative path type, as the meaning of `this` changes in subclasses. The last conditional premise checks that `m` correctly overrides (if it does) the method of the same name in the superclass with the same signature (modulo renaming of type variables).

There are two class typing rules, one for top-level classes and one for nested classes. Both of them are essentially the same: they check that field types and constructor argument types are the same, and that methods are ok in the class. The rule T-TClass for top-level classes also checks that nested classes are ok.

3.4 Operational Semantics

The operational semantics is given by the reduction relation of the form $\mathtt{e} \longrightarrow \mathtt{e}'$, read "expression $\mathtt{e}$ reduces to $\mathtt{e}'$ in one step." We require another lookup function $\mathit{mbody}(\mathtt{m}, \mathtt{A})$, of which we omitted its obvious definition, for the method body with formal parameters, written $\overline{\mathtt{x}}.\mathtt{e}$, of given method and class names.

The reduction rules are given in Figure 6. We write $[\overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{e}/\mathtt{y}]\mathtt{e}_0$ for the expression obtained from $\mathtt{e}_0$ by replacing $\mathtt{x}_1$ with $\mathtt{d}_1$, ..., $\mathtt{x}_n$ with $\mathtt{d}_n$, and $\mathtt{y}$ with $\mathtt{e}$. There are two reduction rules, one for field access and one for method invocation, which are straightforward. The reduction rules may be applied at any

$$\frac{\mathit{fields}(\mathtt{A}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{new\ A(}\overline{\mathtt{e}}\mathtt{).f}_i \longrightarrow \mathtt{e}_i} \qquad \frac{\mathit{mbody}(\mathtt{m<}\overline{\mathtt{P}}\mathtt{>},\ \mathtt{A}) = \overline{\mathtt{x}}\mathtt{.e}_0}{\mathtt{new\ A(}\overline{\mathtt{e}}\mathtt{).<}\overline{\mathtt{P}}\mathtt{>m(}\overline{\mathtt{d}}\mathtt{)} \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ A(}\overline{\mathtt{e}}\mathtt{)/this}]\mathtt{e}_0}$$

Fig. 6. .FJ: Reduction

point in an expression, so we also need the obvious congruence rules (if $\mathtt{e} \longrightarrow \mathtt{e}'$ then $\mathtt{e.f} \longrightarrow \mathtt{e}'\mathtt{.f}$, and the like), omitted here. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

3.5 Type Soundness

The type system is sound with respect to the operational semantics, as expected. Type soundness is proved in the standard manner via subject reduction and progress [22,14]. (Recall that values are defined by: $\mathtt{v} ::= \mathtt{new\ A(}\overline{\mathtt{v}}\mathtt{)}$, where $\overline{\mathtt{v}}$ can be empty.)

Theorem 1 (Subject Reduction). *If* $\Delta;\Gamma \vdash_{\mathtt{A}} \mathtt{e}:\mathtt{T}$ *and* $\mathtt{e} \longrightarrow \mathtt{e}'$*, then* $\Delta;\Gamma \vdash_{\mathtt{A}} \mathtt{e}':\mathtt{T}'$*, for some* $\mathtt{T}'$ *such that* $\Delta \vdash \mathtt{T}'\mathtt{<:}\mathtt{T}$.

Theorem 2 (Progress). *If* $\emptyset;\emptyset \vdash_{\mathtt{B}} \mathtt{e}:\mathtt{A}$ *and* $\mathtt{e}$ *is not a value, then* $\mathtt{e} \longrightarrow \mathtt{e}'$*, for some* $\mathtt{e}'$.

Theorem 3 (Type Soundness). *If* $\emptyset;\emptyset \vdash_{\mathtt{B}} \mathtt{e}:\mathtt{A}$ *and* $\mathtt{e} \longrightarrow^* \mathtt{e}'$ *with* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *is a value* $\mathtt{v}$ *with* $\emptyset;\emptyset \vdash_{\mathtt{B}} \mathtt{v}:\mathtt{A}'$ *and* $\emptyset \vdash \mathtt{A}'\mathtt{<:}\mathtt{A}$.

4 Type Inference for Parametric Method Invocations

The language .FJ in the previous section is considered an intermediate language in which every type argument to parametric methods is made explicit. In this section, we briefly discuss how type arguments can be recovered, give an algorithm for type argument inference, and show its correctness theorem.

The basic idea of type inference is the same as Java 5.0: Given a method invocation expression $\mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)}$ that appears in class $\mathtt{A}$ without specifying type arguments, we can at least compute the type $\mathtt{T}_0$ of $\mathtt{e}_0$, the signature $\mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0$ of the method $\mathtt{m}$, and the types $\overline{\mathtt{T}}$ of (value) arguments. Then, it is easy to see from the rule T-INVK that it suffices to find $\overline{\mathtt{P}}$ that satisfies $\overline{\mathtt{P}}$ <: $\overline{\mathtt{C}}$ and $\overline{\mathtt{T}}$ <: $([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{U}})@\mathtt{T}_0$. In other words, the goal of type inference is to solve the set $\{\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{C}}, \overline{\mathtt{T}}\mathtt{<:}(\overline{\mathtt{U}}@\mathtt{T}_0)\}$ of inequalities with respect to $\overline{\mathtt{X}}$.

We formalize this constraint solving process as function $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S)$. It takes as input a set S of inequalities of the form either $\mathtt{X<:C}$ or $\mathtt{T}_1\mathtt{<:}\mathtt{T}_2$ where $\mathtt{T}_1$ does not contain $\mathtt{X}_i$, and returns a mapping from $\overline{\mathtt{X}}$ to types (more precisely, family names). Δ records other variables' bounds, so $\overline{\mathtt{X}}$ and the domain of Δ are assumed to be disjoint. The definition of $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S)$ is shown in Figure 7. Here, $S_1 \uplus S_2$ is a union of S_1 and S_2, where $S_1 \cup S_2 = \emptyset$. $\mathtt{T}_1 \sqcup_{\Delta} \mathtt{T}_2$ is the least upper bound of $\mathtt{T}_1$ and $\mathtt{T}_2$ (the least upper bound of given two types always exist since we do

$$\begin{array}{ll}
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(\emptyset) & = [\,] \\
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{P.C}\texttt{<:}\mathtt{X}_i\mathtt{.C}\}) & = [\mathtt{X}_i \mapsto \mathtt{P}] \circ \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}([\mathtt{P}/\mathtt{X}_i]S') \\
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{T}_1\texttt{<:}\mathtt{X}_i\} \uplus \{\mathtt{T}_2\texttt{<:}\mathtt{X}_i\}) & = \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{(\mathtt{T}_1 \sqcup_{\Delta} \mathtt{T}_2)\texttt{<:}\mathtt{X}_i\}) \\
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{T}\texttt{<:}\mathtt{X}_i, \mathtt{X}_i\texttt{<:}\mathtt{C}_i\}) & = \begin{cases} [\mathtt{X}_i \mapsto \mathtt{T}] \circ \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S') & \text{if } \Delta \vdash \mathtt{T}\texttt{<:}\mathtt{C}_i \text{ and } \mathtt{X}_i \notin S' \\ \mathit{fail} & \text{otherwise} \end{cases} \\
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{X}_i\texttt{<:}\mathtt{C}_i\}) & = \begin{cases} [\mathtt{X}_i \mapsto \mathtt{C}_i] \circ \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S') & \text{if } \mathtt{X}_i \notin S' \\ \mathit{fail} & \text{otherwise} \end{cases} \\
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{T}_1\texttt{<:}\mathtt{T}_2\}) & = \begin{cases} \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S') & \text{if } \Delta \vdash \mathtt{T}_1\texttt{<:}\mathtt{T}_2 \\ \mathit{fail} & \text{otherwise} \end{cases}
\end{array}$$

Fig. 7. Algorithm for Type Argument Synthesis

not have interfaces, which can extend more than one interface.) We assume that each clause is applied in the order shown—thus, for example, the fourth clause will not be applied until there is only one inequation of the form $\mathtt{T}\texttt{<:}\mathtt{X}_i$.

The algorithm is explained as follows. The second clause is the case where a formal argument type is $\mathtt{X}_i\mathtt{.C}$ and the corresponding actual is `P.C`: since `P.C` has only a trivial supertype (namely, itself), $\mathtt{X}_i$ must be `P`. The third clause is the case where a type variable has more than one lower bound: we replace two inequalities by one using the least upper bound. The following two clauses are applied when no other constraints on $\mathtt{X}_i$ appear elsewhere; it checks whether the constraint is satisfiable.

Now, we state the theorem of correctness of type inference. It means that, if type inference succeeds, it gives the least type arguments.

Theorem 4 (Type Inference Correctness). *If* $\Delta; \Gamma \vdash_{\mathtt{A}} \mathtt{e}_0 \mathtt{:} \mathtt{T}_0$ *and* $\mathit{mtype}(\mathtt{m}, \Delta(\mathtt{T}_0)\mathtt{@A}) = \mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U}_0$ *and* $\Delta; \Gamma \vdash_{\mathtt{A}} \overline{\mathtt{e}} \mathtt{:} \overline{\mathtt{T}}$ *and* $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(\{\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{C}}, \overline{\mathtt{T}}\texttt{<:}(\overline{\mathtt{U}}\mathtt{@}\mathtt{T}_0)\})$ *returns* $\sigma = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{P}}]$*, then* $\Delta; \Gamma \vdash_{\mathtt{A}} \mathtt{e}_0\mathtt{.<}\sigma\overline{\mathtt{X}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} : (\sigma\mathtt{U}_0)\mathtt{@}\mathtt{T}_0$*. Moreover,* σ *is the least solution if every* $\mathtt{X}_i$ *occurs in* $\overline{\mathtt{U}}$ *in the sense that for any* σ' *such that* $\Delta; \Gamma \vdash_{\mathtt{A}} \mathtt{e}_0\mathtt{.<}\sigma'\overline{\mathtt{X}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} : (\sigma'\mathtt{U}_0)\mathtt{@}\mathtt{T}_0$*, it holds that* $\Delta \vdash \sigma(\mathtt{X}_i) \texttt{<:} \sigma'(\mathtt{X}_i)$ *for any* $\mathtt{X}_i$*.*

5 Related Work

As we have already mentioned, in the original formulation of family polymorphism [11] nested classes are members (or attributes) of an object of their enclosing class. Thus, to create node or edge objects, one first has to instantiate `Graph` and then to invoke `new` on a class attribute of that object. It would be written as

```
Graph g = new Graph();
g.Node n = new g.Node(.. );  g.Edge e = new g.Edge(.. );
```

Notice that the types of nodes and edges would include a reference `g` to the `Graph` object. Relative path types `.Node` and `.Edge` would respectively become this.Node and this.Edge, where the meaning of types changes as the meaning of this changes due to usual late-binding. Finally, connectAll() would take four *value* arguments instead of one type and three value arguments as:

```
void connectAll(Graph g,
                g.Edge[] es, g.Node n1, g.Node n2){ .. }
```

Notice that the first argument appears as part of types of the following arguments; it is required for a type system to guarantee that es, n1, and n2 belong to the same graph. As a result, a type system is equipped with dependent types, such as g.Edge, which can be quite tricky (especially in the presence of side-effects). We deliberately avoid such types by identifying families with classes. As a byproduct, as shown in the previous section, we have discovered that GJ-style type inference is easy to extend to this setting. Although complex, the original approach has one apparent advantage: one can instantiate arbitrary number of Graph objects and distinguish nodes and edges of different graphs by static types. Scala [19] also support family polymorphism, based on dependent types.

Historically, the mismatching problem of recursive class definitions has been studied in the context of binary methods [4], which take an object of the same class as the receiver, hence the interface is (self-)recursive. In particular, Bruce's series of work [7,5] introduced the notion of *MyType* (or sometimes called *ThisType*), which is the type of this and changes its meaning along the inheritance chain, just as our relative path types. Later, he extended the idea to mutually recursive type/class definitions [6,8,3] by introducing constructs to group mutually recursive definitions, and the notion of *MyGroup*, which is a straightforward extension of *MyType* to the mutually recursive setting. Jolly et al. [15] has designed the language called Concord by following this approach and has applied to a Java-like language with a name-based type system. The core type system has been proven sound. Our approach is similar to them in the sense that dependent types are not used. However, in these work, family-polymorphic methods are not taken into account very seriously, although a similar idea is mentioned in Bruce et al. [6] and it can be considered a generalization of match-bound polymorphic methods in the language $\mathcal{LOOM}$ [7]. In Bruce et al. [6], inheritance is considered subtyping, so ColorWeightGraph.Node <: Graph.Node, for example. To ensure type safety, they introduced the notion of exact types and allow to invoke a method that take an argument of the same family only when the receiver's family is *exactly* known. We have avoided them by viewing every (nested-class) type as exact. JX [17], an extension of Java with nested inheritance, supports a similar programming style but exactness is kept track of by using not types but Java's final.

In Concord, gbeta, Scala, and JX, an inheritance relation between nested classes can be introduced. For example, C.F can be a subclass of C.E and, in a subclass D of C, the relationship is preserved while members can be added to both E and F. Although useful, we have carefully avoided this feature, too, which is not strongly required by family polymorphism, since there is a semantic complication as in languages with multiple inheritance: D.F may inherit conflicting members of the same name from C.F and D.E.

Finally, we should note that programming described in Section 2 could be carried out in Java 5.0 proper, which is equipped with generics [2] and F-bounded polymorphism [9], by using the technique [21] used to solve the "expression problem". It requires, however, a lot of boilerplate code for type parameterization, which makes programs less easy to grasp.

6 Concluding Remarks

We have identified a minimal set of language features to solve the problem of mismatching between mutually recursive classes and inheritance. Our proposal is lightweight in the sense that the type system, which avoids (value) dependent types, is much simpler than the original formulation of family polymorphism and easy to apply to mainstream languages such as Java and C$^{\#}$. We have shown type safety of the language mechanism by proving a type soundness theorem for the formalized core language .FJ. We have also formalized an algorithm for type argument inference for family polymorphic methods with its correctness theorem. Although .FJ is not equipped with generics, we believe it can be easily integrated into the ordinary type argument inference algorithm.

A prototype compiler for the language features presented is being implemented on top of `javac`. As in the implementation of Java generics, new features are compiled by *erasure* [2,14] to Java proper—relative path types are translated to ordinary static nested class types with the insertion of typecasts (guaranteed by the type system to succeed) where necessary to make the translated program well typed.

We feel that the principle of classes-as-families is worth pursuing. There have been advanced work based on object-based families, such as higher-order hierarchies [12]. It is interesting to investigate whether this principle can be applied to those advanced ideas.

Acknowledgements. Part of the development of the prototype compiler was done by Stefano Olivieri and Mirko Viroli during their visit to Kyoto University. Comments from anonymous referees helped up improve the final presentation. The first author would like to thank members of the Kumiki project for fruitful discussions on this subject. This work was supported in part by Grant-in-Aid for Scientific Research on Priority Areas Research No. 13224013 from MEXT of Japan (Igarashi), and from the Italian PRIN 2004 Project "Extensible Object Systems" (Viroli).

References

1. David Aspinall and Martin Hofmann. Dependent types. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 2, pages 45–86. The MIT Press, 2005.
2. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of OOPSLA'98*, pages 183–200, Vancouver, BC, October 1998.

3. Kim B. Bruce. Some challenging typing issues in object-oriented languages. In *Proceedings of Workshop on Object-Oriented Development (WOOD'03)*, volume 82 of *Electronic Notes in Theoretical Computer Science*, 2003.
4. Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary method. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.
5. Kim B. Bruce and J. Nathan Foster. LOOJ: Weaving LOOM into Java. In *Proceedings of ECOOP2004*, Springer LNCS 3086, Oslo, Norway, June 2004.
6. Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Proceedings of ECOOP'98*, Springer LNCS 1445, pages 523–549, Brussels, Belgium, July 1998.
7. Kim B. Bruce, Leaf Petersen, and Adrian Fiech. Subtyping is not a good "match" for object-oriented languages. In *Proceedings of ECOOP'97*, Springer LNCS 1241, pages 104–127, Jyväskylä, Finland, June 1997.
8. Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Proceedings of 15th Conference on the Mathematical Foundations of Programming Semantics (MFPS XV)*, ENTCS 20, New Orleans, LA, April 1999. Elsevier.
9. Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of FPCA'89*, pages 273–280, London, England, September 1989. ACM Press.
10. Erik Ernst. *gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance.* PhD thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1999.
11. Erik Ernst. Family polymorphism. In *Proceedings of ECOOP2001*, Springer LNCS 2072, pages 303–326, Budapest, Hungary, June 2001.
12. Erik Ernst. Higher-order hierarchies. In *Proceedings of ECOOP2003*, Springer LNCS 2743, pages 303–328, Darmstadt, Germany, July 2003.
13. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of POPL'98*, pages 171–183, San Diego, CA, January 1998.
14. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.
15. Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types: Concord. In *Proceedings of 6th ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP2004)*, June 2004.
16. Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA'89*, pages 397–406, October 1989.
17. Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proceedings of OOPSLA'04*, October 2004.
18. Martin Odersky. Inferred type instantiation for GJ. Available at `http://lampwww.epfl.ch/~odersky/papers/localti02.html`, January 2002.
19. Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings of ECOOP'03*, Springer LNCS 2743, pages 201–224, Darmstadt, Germany, July 2003.
20. Kresten Krab Thorup and Mads Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In *Proceedings of ECOOP'99*, Springer LNCS 1628, pages 186–204, Lisbon, Portugal, June 1999.

21. Mads Torgersen. The expression problem revisited: Four new solutions using generics. In *Proceedings of ECOOP2004*, Springer LNCS 3086, pages 123–146, Oslo, Norway, June 2004.
22. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

A Portable and Customizable Profiling Framework for Java Based on Bytecode Instruction Counting

Walter Binder

Ecole Polytechnique Fédérale de Lausanne (EPFL),
Artificial Intelligence Laboratory,
CH-1015 Lausanne, Switzerland
walter.binder@epfl.ch

Abstract. Prevailing profilers for Java, which rely on standard, native-code profiling interfaces, are not portable, give imprecise results due to serious measurement perturbation, and cause excessive overheads. In contrast, program transformations allow to generate reproducible profiles in a fully portable way with significantly less overhead. This paper presents a profiling framework that instruments Java programs at the bytecode level to build context-sensitive execution profiles at runtime. The profiling framework includes an exact profiler as well as a sampling profiler. User-defined profiling agents can be written in pure Java, too, in order to customize the runtime processing of profiling data.

Keywords: Profiling, program transformations, bytecode instrumentation, dynamic metrics, Java, JVM

1 Introduction

Most prevailing Java profilers rely on the Java Virtual Machine Profiling Interface (JVMPI) [14,15] or on the JVM Tool Interface (JVMTI) [16], which provide a set of hooks to the Java Virtual Machine (JVM) to signal interesting events, such as thread start and object allocations. Usually, such profilers can operate in two modes: In the *exact profiling mode*, they track each method invocation, whereas in the *sampling mode*, the profiler spends most of the time sleeping and periodically (e.g., every few milliseconds) wakes up to register the current stack trace.

Profilers based on the JVMPI or JVMTI interfaces implement profiling agents to intercept various events, such as method invocations. Unfortunately, these profiling agents have to be written in platform-dependent native code, contradicting the Java motto 'write once and run anywhere'. Because exact profiling based on these APIs may cause an enormous overhead (in extreme cases we even experienced a slowdown of factor 4 000 and more), developers frequently resort to more efficient sampling-based profilers to analyze the performance of complex system, such as application servers. Many prevailing sampling profilers for Java use an external timer to trigger the sampling, resulting in non-deterministic behaviour: For the same program and input, the generated profiles may vary very much depending on the processor speed and the system load. In many cases the accuracy of the resulting profiles is so low that a reasonable performance analysis based on these profiles is not possible. In short, most prevailing

Java profilers are either too slow or too imprecise to generate meaningful profiles for complex systems.

To solve these problems, we developed a novel profiling framework that relies neither on the JVMPI nor on the JVMTI, but directly instruments the bytecode of Java programs in order to create a profile at runtime. Our framework includes the exact profiler JP as well as the sampling profiler Komorium. Both profilers exploit the number of executed bytecodes[1] as platform-independent profiling metric, enabling reproducible profiles and a fully portable implementation of the profilers. Moreover, user-defined profiling agents written in pure Java may customize the profile generation.

As contributions, this paper introduces bytecode counting as profiling metric and defines an innovative profiling framework that is completely based on program transformations. Two profiling schemes are presented, one for exact profiling and one for sampling profiling. We implemented and evaluated both of them: The exact profiler JP causes 1–2 orders of magnitude less overhead than prevailing exact profilers. The sampling profiler Komorium causes less overhead than existing sampling profilers, and the resulting profiles are much more accurate.

The remainder of this paper is structured as follows: Section 2 argues for bytecode counting as profiling metric. Section 3 introduces the data structures on which our profiling framework is based. In Section 4 we explain the program transformation scheme underlying the exact profiler JP. Section 5 discusses the sampling profiler Komorium. In Section 6 we evaluate the performance of our profilers, as well as the accuracy of the profiles generated by the sampling profiler. Section 7 summarizes the limitations of our approach and outlines some ideas for future improvements. Finally, Section 8 discusses related work and Section 9 concludes this paper.

2 Bytecode Counting

Most existing profilers measure the CPU consumption of programs in seconds. Although the CPU second is the most common profiling metric, it has several drawbacks: It is platform-dependent (for the same program and input, the CPU time differs depending on hardware, operating system, and JVM), measuring it accurately may require platform-specific features (such as special operating system functions) limiting the portability of the profilers, and results may not be easily reproducible (the CPU time may depend on factors such as system load). Furthermore, measurement perturbation is often a serious problem: The measured CPU consumption of the profiled program may significantly differ from the effective CPU consumption when the program is executed without profiling. The last point is particularly true on JVMs where the use of the JVMPI disables just-in-time compilation.

For these reasons, we follow a different approach, using the number of executed bytecodes as profiling metric, which has the following benefits:

– **Platform-independ profiles:** The number of executed bytecodes is a platform-independent metric [10]. Although the CPU time of a deterministic program with a given input varies very much depending on the performance of the underlying

[1] In this paper 'bytecode' stands for 'JVM bytecode instruction'.

hardware and virtual machine (e.g., interpretation versus just-in-time compilation), the number of bytecodes issued by the program remains the same, independent of hardware and virtual machine implementation (assuming the same Java class library is used).

- **Reproducible profiles:** For deterministic programs, the generated profiles are completely reproducible.
- **Comparable profiles:** Profiles collected in different environments are directly comparable, since they are based on the same platform-independent metric.
- **Accurate profiles:** The profile reflects the number of bytecodes that a program would execute without profiling, i.e., the profiling itself does not affect the generated profile (no measurement perturbation).
- **Portable and compatible profiling scheme:** Because counting the number of executed bytecodes does not require any hardware- or operating system-specific support, it can be implemented in a fully portable way. The profiling scheme is compatible also with JVMs that support neither the JVMPI nor the JVMTI, or that provide limited support for profiling in general.
- **Portable profiling agents:** Custom profiling agents can be written in pure Java and are better integrated with the environment. Hence, profiling agents are portable and can be used in all kinds of JVMs.
- **Flexible profiling agents**: Profiling agents can be programmed to preserve a trace of the full call stack, or to compact it at certain intervals, whereas existing profilers frequently only support a fixed maximal stack depth.
- **Fine-grained control of profiling agent activation:** Profiling agents are invoked in a deterministic way by each thread after the execution of a certain number of bytecodes, which we call the *profiling granularity*. Profiling agents can dynamically adjust the profiling granularity in a fine-grained way.
- **Reduced overhead:** The overhead is rather low compared to classical approaches, since it does not prevent the underlying JVM from putting all its optimization facilities to work during the profiling.

Consequently, bytecode counting is key to the provision of a new class of portable, platform-independent profiling tools, which gives advantages to the tool users as well as to the tool implementors:

On the one hand, bytecode counting eases profiling, because thanks to the platform-independence of this metric [10], the concrete environment is not of importance. Thus, the developer may profile programs in the environment of his preference. Since factors such as the system load do not affect the profiling results, the profiler may be executed as a background process on the developer's machine. This increases productivity, as there is no need to set up and maintain a dedicated, 'standardized' profiling environment.

On the other hand, bytecode counting enables fully portable profiling tools. This helps to reduce the development and maintenance costs for profiling tools, as a single version of a profiling tool can be compatible with any kind of virtual machine. This is in contrast to prevailing profiling tools, which exploit low-level, platform-dependent features (e.g., to obtain the exact CPU time of a thread from the underlying operating system) and require profiling agents to be written in native code.

- `createMID(STRING` $class$`, STRING` $name$`, STRING` sig`): MID`
  Creates a new method identifier, consisting of class name, method name, and method signature.
- `getClass(MID` mid`): STRING`
  Returns the class name of mid.
  `getClass(createMID(`c`,` x`,` y`))` $= c$.
- `getName(MID` mid`): STRING`
  Returns the method name of mid.
  `getName(createMID(`x`,` n`,` y`))` $= n$.
- `getSig(MID` mid`): STRING`
  Returns the method signature of mid.
  `getSig(createMID(`x`,` y`,` s`))` $= s$.

Fig. 1. Method identifier `MID`

- `getOrCreateRoot(THREAD` t`): IC`
  Returns the root node of a thread's MCT. If it does not exist, it is created.
- `profileCall(IC` $caller$`, MID` $callee$`): IC`
  Registers a method invocation in the MCT. The returned IC represents the callee method, identified by $callee$. It is a child node of $caller$ in the MCT.
- `getCaller(IC` $callee$`): IC`
  Returns the caller `IC` of $callee$. It is the parent node of $callee$ in the MCT.
  `getCaller(profileCall(`c`,` x`))` $= c$.
  This operation is not defined for the root of the MCT.
- `getCalls(IC` c`): INT`
  Returns the number of invocations of the method identified by `getMID(`c`)` with the caller `getCaller(`c`)`.
  `getCalls(profileCall(`x`,` y`))` ≥ 1.
  This operation is not defined for the root of the MCT.
- `getMID(IC` c`): MID`
  Returns the method identifier associated with c.
  `getMID(profileCall(`x`,` $callee$`))` $= callee$.
  This operation is not defined for the root of the MCT.
- `getCallees(IC` c`): SET OF IC`
  Returns the set of callee `IC`s of c.
  $\forall x \in$ `getCallees(`c`):getCaller(`x`)` $= c$.
  $\forall x \in$ `getCallees(`c`):getCalls(`x`)` ≥ 1.
- `profileInstr(IC` ic`, INT` $bytecodes$`): IC`
  Registers the execution of a certain number of bytecodes in ic. The bytecode counter in ic is incremented by $bytecodes$. Returns ic, after its bytecode counter has been updated. This operation is not defined for the root of the MCT.
- `getInstr(IC` ic`): INT`
  Returns the number of bytecodes executed in ic.
  `getInstr(profileInstr(`x`,` b`))` $\geq b$.
  This operation is not defined for the root of the MCT.

Fig. 2. Method invocation context `IC`

3 Profiling Data Structures

In this section we define the data structures used by our profiling framework as abstract datatypes. A detailed presentation of the Java implementation of these data structures had to be omitted due to space limitations.

3.1 Method Call Tree (MCT)

For exact profiling, we rewrite JVM bytecode in order to create a Method Call Tree (MCT), where each node represents all invocations of a particular method with the same call stack. The parent node in the MCT corresponds to the caller, the children nodes correspond to the callees. The root of the MCT represents the caller of the main method. With the exception of the root node, each node in the MCT stores profiling information for all invocations of the corresponding method with the same call stack. Concretely, it stores the number of method invocations as well as the number of bytecodes executed in the corresponding calling context, excluding the number of bytecodes executed by callee methods (each callee has its own node in the MCT).

- `getOrCreateAC(THREAD` t`): AC`
  Returns the activation counter of a thread. If it does not exist, it is created.
- `setValue(AC` ac`, INT` v`): AC`
  Returns ac, after its value has been updated to v.
- `getValue(AC` ac`): INT`
  Returns the value of ac.
  `getValue(setValue(`x`,` v`)) =` v.

Fig. 3. Activation counter `AC`

In order to prevent race conditions, either access to the MCT has to be synchronized, or each thread has to maintain its own copy of the tree. To avoid expensive synchronization and to allow profiling agents to keep the profiling statistics of different threads separately, we chose to create a separate MCT for each thread in the system.[2]

The MCT is similar to the Calling Context Tree (CCT) [1]. However, in contrast to the CCT, the depth of the MCT is unbounded. Therefore, the MCT may consume a significant amount of memory in the case of very deep recursions. Nonetheless, for most programs this is not a problem: According to Ball and Larus [5], path profiling (i.e., preserving exact execution history) is feasible for a large portion of programs.

We define two abstract datatypes to represent a MCT, the method identifier `MID` (see Fig. 1) and the method invocation context `IC` (see Fig. 2). A method invocation context is a node in the MCT, encapsulating a method invocation counter and a bytecode counter. We assume the existence of the types `INT`, `STRING`, and `THREAD`, as well as the possibility to create aggregate types (`SET OF`).

3.2 Activation Counter

In order to schedule the regular activation of a user-defined profiling agent in a platform-independent way, our profilers maintain a counter of the (approximate) number of executed bytecodes for each thread. If this counter exceeds the current profiling granularity, the profiling agent is invoked in order to process the collected execution statistics. The abstract datatype `AC` (see Fig. 3) represents an activation counter for each thread.

4 Exact Profiling

In this section we describe a fully portable scheme for exact profiling. In order to validate our approach, we implemented the exact profiler JP, which is compatible with standard JVMs. JP relies neither on the JVMPI nor on the JVMTI, but directly instruments the bytecode of Java programs in order to obtain detailed execution statistics.

In Section 4.1 we explain how programs are transformed to create MCTs at runtime. While Section 4.2 discusses the necessary code instrumentation to maintain the bytecode counters within the MCTs, Section 4.3 explicates the periodic activation of a custom profiling agent. Finally, in Section 4.4 we illustrate the program transformations with an example.

[2] At the implementation level, a thread-local variable may be used to store a reference to the root of a thread's MCT. Each thread gets its own instance of the thread-local variable. In Java, thread-local variables are instances of `java.lang.ThreadLocal`.

4.1 MCT Creation

JP rewrites JVM bytecode in order to pass the method invocation context ic_{caller} (type `IC`) of the caller as an extra argument to the callee method (i.e., JP extends the signatures of all non-native methods with the additional argument). In the beginning of a method[3] identified by mid_{callee} (type `MID`), the callee executes a statement corresponding to

ic_{callee} = `profileCall(`ic_{caller}`,` mid_{callee}`);`

in order to obtain its own (i.e., the callee's) method invocation context ic_{callee}.

Because native code is not changed by the rewriting, JP adds simple wrapper methods with the unmodified signatures which obtain the current thread's MCT root by calling `getOrCreateRoot(`t`)`, where t represents the current thread. Therefore, native code is able to invoke Java methods with the unmodified signatures.[4]

For each Java method, we add a static field to hold the corresponding method identifier. In the static initializer we call `createMID(`$classname$`,` $methodname$`,` $signature$`)` in order to allocate a method identifier for each Java method.

4.2 Bytecode Counting

For each method invocation context ic, JP computes the number of executed bytecodes. JP instruments the bytecode of methods in order to invoke `profileInstr(`ic`,` $bytecodes$`)` according to the number of executed bytecodes. For each Java method, JP performs a basic block analysis (BBA) to compute a control flow graph. In the beginning of each basic block it inserts a code sequence that implements this update of the bytecode counter.

The BBA algorithm is not hard-coded in JP, via a system property the user can specify a custom analysis algorithm. JP itself offers two built-in BBA algorithms, which we call 'Default BBA' resp. 'Precise BBA'. In the 'Default BBA', only bytecodes that may change the control flow non-sequentially (i.e., jumps, branches, return of method or JVM subroutine, exception throwing) end a basic block. Method or JVM subroutine invocations do not end basic blocks of code, because we assume that the execution will return after the call. This definition of basic block corresponds to the one used in [7] and is related to the factored control flow graph [8].

The advantage of the 'Default BBA' is that it creates rather large basic blocks. Therefore, the number of locations is reduced where updates to the bytecode counter have to be inserted, resulting in a lower profiling overhead. As long as no exceptions are thrown, the resulting profiling information is precise. However, exceptions (e.g., an invoked method may terminate abnormally throwing an exception) may cause some imprecision in the accounting, as we always count all bytecodes in a basic block, even

[3] In this paper we do not distinguish between Java methods and constructors, i.e., 'method' stands for 'method or constructor'.

[4] For native methods, which we cannot rewrite, we add so-called 'reverse' wrappers which discard the extra `IC` argument before invoking the native method. The 'reverse' wrappers allow rewritten code to invoke all methods with the additional argument, no matter whether the callee is native or not.

- `register(THREAD t, IC root): INT`
  This operation is invoked whenever a new thread t is created. It is called by `getOrCreateRoot(t)`, if a new MCT root node ($root$) has been allocated. For each thread, this operation is invoked only once, when it starts executing instrumented code (a wrapper method as discussed in Section 4.1). After `register(t, root)` has been called, the profiling agent must be prepared to handle subsequent invocations of `processMCT(IC)` by the thread t.
  `register(t, root)` returns the current profiling granularity for t, i.e., the approximate number of bytecodes to execute until t will invoke `processMCT(IC)` for the first time.
- `processMCT(IC ic): INT`
  This operation is periodically invoked by each thread in the system. Whenever `processMCT(ic)` is called, the profiling agent has to process the current thread's MCT. ic is the method invocation context corresponding to the method that is currently being executed. The profiling agent may obtain the root of the current thread's MCT either from a map (to be updated upon invocations of `register(THREAD, IC)`) or by successively applying `getCaller(IC)`. `processMCT(IC)` allows the profiling agent to integrate the MCTs of the different threads into a global MCT, or to generate continuous metrics [10], which is particularly useful to display up-to-date profiling information of long running programs, such as application servers.
  `processMCT(IC)` returns the current profiling granularity for the calling thread, i.e., the approximate number of bytecodes to execute until the current thread will invoke `processMCT(IC)` again.

Fig. 4. `ExactProfiler`

though some of them may not be executed in case of an exception. I.e., using the 'Default BBA', we may count more bytecodes than are executed.

If the user wants to avoid this potential imprecision, he may select the 'Precise BBA', which ends a basic block after each bytecode that either may change the control flow non-sequentially (as before), or may throw an exception. As there are many bytecodes that may throw an exception (e.g., `NullPointerException` may be raised by most bytecodes that require an object reference), the resulting average basic block size is smaller. This inevitably results in a higher overhead for bytecode counting, because each basic block is instrumened by JP.

4.3 Periodic Activation of Custom Profiling Agents

JP supports user-defined profiling agents which are periodically invoked by each thread in order to aggregate and process the MCT collected by the thread. The custom profiling agent has to implement the abstract datatype `ExactProfiler` (see Fig. 4).

Each thread maintains an activation counter ac (type `AC`) in order to schedule the regular activation of the custom profiling agent. The value of ac is an upper bound of the number of executed bytecodes since the last invocation of `processMCT(IC)`. In order to make ac directly accessible within each method, we pass it as an additional argument to all invocations of non-native methods. If the value of ac exceeds the profiling granularity, the thread calls `processMCT(IC)` of the profiling agent. Note that the value of ac is not part of the profiling statistics, it is only used at runtime to ensure the periodic activation of the profiling agent.

The value of ac runs from the profiling granularity down to zero, because there are dedicated bytecodes for the comparison with zero. I.e., the following conditional is used to schedule the periodic activation of the profiling agent and to reset ac (ic refers to the current method invocation context):

```
if (getValue(ac) <= 0) setValue(ac, processMCT(ic));
```

The updates of ac are correlated to the updates of the bytecode counters within the MCT (`profileInstr(IC, INT)`). However, in order to reduce the overhead, the value of ac is not updated in every basic block of code, but only in the beginning of each method, exception handler, and JVM subroutine, as well as in the beginning of each loop. Each time it is decremented by the number of bytecodes on the longest execution path until the next update or until the method terminates. This ensures that the value of ac is an upper bound of the number of executed bytecodes.

The conditional that checks whether `processMCT(IC)` has to be called is inserted in the beginning of each method and in each loop, in order to ensure its presence in recursions and iteration. As an optimization, we omit the conditional in the beginning of a method, if before invoking any method, each execution path either terminates or passes by an otherwise inserted conditional. For instance, this optimization allows to remove the check in the beginning of leaf methods.

4.4 Rewriting Example

The example in Fig. 5 illustrates the program transformations performed by JP: To the left is the class `Foo` with the method `sum(int, int)` before rewriting, to the right is the rewritten version.[5] `sum(int, int)` computes the following mathematical function: $sum(a, b) = \sum_{i=a}^{b} f(i)$. The method `int f(int)`, which is not shown in Fig. 5, is transformed in a similar way as `sum(int, int)`. In `sum(int, int)` we use an infinite `while()` loop with an explicit conditional to end the loop instead of a `for()` loop that the reader might expect, in order to better reflect the basic block structure of the compiled JVM bytecode.

For this example, we used the 'Default BBA' introduced in Section 4.2. `sum(int, int)` has 4 basic blocks of code: The first one (2 bytecodes) initializes the local variable `result` with zero, the second one (3 bytecodes) compares the values of the local variables `from` and `to` and branches, the third one (2 bytecodes) returns the value of the local variable `result`, and the fourth block (7 bytecodes) adds the return value of `f(from)` to the local variable `result`, increments the local variable `from`, and jumps to the begin of the loop.

In the rewritten code, the static initializer allocates the method identifier `mid_sum` to represent invocations of `sum(int, int)` in the MCT. The rewritten method receives 2 extra arguments, the activation counter (type `AC`) and the caller's method invocation context (type `IC`). First, the rewritten method updates the MCT and obtains its own (the callee's) method invocation context (`profileCall(IC, MID)`). The bytecode counter within the callee's method invocation context is incremented in the beginning of each basic block of code by the number of bytecodes in the block (`profileInstr(IC, INT)`).

The activation counter `ac` is updated in the beginning of the method and in the loop. It is reduced by the number of bytecodes on the longest execution path until the next

[5] For the sake of better readability, in this paper we show all transformations on Java-based pseudo-code, whereas our profiler implementations work at the JVM bytecode level. The operations on the abstract datatypes `MID`, `IC`, `AC`, and `ExactProfiler` are directly inlined in order to simplify the presentation.

```
class Foo {                    class Foo {
                                  private static final MID mid_sum;
                                  static {
                                     String cl = Class.forName("Foo").getName();
                                     mid_sum   = createMID(cl, "sum", "(II)I");
                                  }

   static int sum(int from,       static int sum(int from,
                  int to) {                      int to, AC ac, IC ic) {
                                     ic = profileCall(ic, mid_sum);
                                     profileInstr(ic, 2);
                                     setValue(ac, getValue(ac) - 2);
      int result = 0;                int result = 0;
      while (true) {                 while (true) {
                                        profileInstr(ic, 3);
                                        setValue(ac, getValue(ac) - 10);
                                        if (getValue(ac) <= 0)
                                           setValue(ac, processMCT(ic));
         if (from > to) {               if (from > to) {
                                           profileInstr(ic, 2);
            return result;                 return result;
         }                              }
                                        profileInstr(ic, 7);
         result += f(from);             result += f(from, ac, ic);
         ++from;                        ++from;
      }                              }
   }                              }

                                  static int sum(int from, int to) {
                                     Thread t = Thread.currentThread();
                                     return sum(from, to, getOrCreateAC(t),
                                                   getOrCreateRoot(t));
                                  }
}                              }
```

Fig. 5. Rewriting example: Program transformations for exact profiling

update or method termination. For instance, in the loop it is incremented by 10 $(3 + 7)$, as this is the length of the execution path if the loop is repeated. The other path, which returns, executes only 5 bytecodes $(3 + 2)$. The conditional is present in the loop, but not in the beginning of the method, since the only possible execution path passes by the conditional in the loop before invoking any method.

A wrapper method with the unmodified signature is added to allow native code, which is not aware of the additional arguments, to invoke the rewritten method. The wrapper method obtains the current thread's activation counter as well as the root of its MCT before invoking the instrumented method with the extra arguments.

5 Sampling-Based Profiling

Even though exact profiling based on the program transformation scheme presented in Section 4 causes considerably less overhead than prevailing exact profilers, the overhead may still be too high for complex applications. Moreover, for applications with many concurrent threads, maintaining a separate MCT for each thread may consume a large amount of memory. For these reasons, we developed the sampling profiler Komorium, which is also based on program instrumentation. Komorium relies on the periodic activation of a user-defined profiling agent to process samples of the call stack.

Sampling profiles are different from exact ones, since they expose neither the absolute number of method invocations nor the absolute number of executed bytecodes. However, sampling profiles can be used to estimate the relative distribution of processing effort, guiding the developer in which parts program optimizations may pay off. In general, there is a correlation between the number of times a certain call stack is reported to the profiling agent and the amount of processing spent in the corresponding calling context. Typically, the profiling agent counts the number of occurrences of the different samples (call stacks). The relative frequency of a certain call stack S (i.e., $\frac{\text{number of occurrences of } S}{\text{total number of samples}}$) approximates the proportion of processing spent in the corresponding calling context.

5.1 Call Stack Reification

In order to make the call stack available at execution time, Komorium reifies the current call stack as a pair $\langle mids, sp \rangle$, where $mids$ is an array of method identifiers (type `MID[]`) and sp is the stack pointer (type `INT`), which denotes the next free element on the reified stack. $mids[i]$ are the identifiers of the activated methods ($0 \leq i < sp$). $mids[0]$ is the bottom of the reified stack and $mids[sp-1]$ is its top, corresponding to the currently executing method.

Komorium transforms programs in order to pass the reified call stack of the caller as 2 extra arguments to the callee method (i.e., Komorium extends the signatures of all non-native methods with the additional arguments). In the beginning of a method identified by mid, the callee executes a statement corresponding to

$mids$`[`sp`++] = `mid`;`

in order to push its method identifier onto the reified stack. The integer sp is always passed by value. I.e., callees receive (a copy of) the new value of sp and may increment it, which does not affect the value of sp in the caller. If a method `m()` invokes first `a()` and then `b()`, both `a()` and `b()` will receive the same $mids$ reference and the same value of sp as extra arguments. `b()` will overwrite the method identifiers that were pushed onto the reified stack during the execution of `a()`.

Compatibility with native code is achieved in a similar way as explained in Section 4.1: Because native code is not changed by the rewriting, Komorium adds simple wrapper methods with the unmodified signatures which allocate an array to represent the reified stack. The initial value of the stack pointer is zero.

5.2 Periodic Sampling

The custom profiling agent has to implement the abstract datatype `SamplingProfiler` (see Fig. 6). In order to schedule the regular activation of the custom profiling agent, each thread maintains an activation counter ac, in a similar way as described in Section 4.3. ac is updated in the beginning of each basic block of code; the basic blocks are computed by the 'Default BBA' introduced in Section 4.2. The conditional that checks whether `processSample(MID[], INT)` has to be invoked is inserted in each basic block after the update of ac. In contrast to the exact profiler, we do not reduce overhead by computing an upper bound of the number

- `register(THREAD` t`): INT`
  This operation is invoked whenever a new thread t is created. After `register(`t`)` has been called, the profiling agent must be prepared to handle subsequent invocations of `processSample(MID[], INT)` by t. `register(`t`)` returns the current profiling granularity for t.
- `processSample(MID[]` $mids$`,`
  `INT` sp`): INT`
  This operation is periodically invoked by each thread in the system in order to process a sample of the reified call stack, which is represented by the pair $\langle mids, sp\rangle$. `processSample(MID[], INT)` returns the current profiling granularity for the calling thread.

Fig. 6. `SamplingProfiler`

```
class Foo {                class Foo {
                              private static final MID mid_sum;
                              static {
                                 String cl = Class.forName("Foo").getName();
                                 mid_sum   = createMID(cl, "sum", "(II)I");
                              }

   static int sum(int from,   static int sum(int from,
                  int to) {                  int to, AC ac, MID[] mids, int sp) {
                                 mids[sp++] = mid_sum;
                                 setValue(ac, getValue(ac) - 2);
                                 if (getValue(ac) <= 0)
                                    setValue(ac, processSample(mids, sp));
      int result = 0;            int result = 0;
      while (true) {             while (true) {
                                    setValue(ac, getValue(ac) - 3);
                                    if (getValue(ac) <= 0)
                                       setValue(ac, processSample(mids, sp));
         if (from > to) {           if (from > to) {
                                       setValue(ac, getValue(ac) - 2);
                                       if (getValue(ac) <= 0)
                                          setValue(ac, processSample(mids, sp));
             return result;            return result;
         }                          }
                                    setValue(ac, getValue(ac) - 7);
                                    if (getValue(ac) <= 0)
                                       setValue(ac, processSample(mids, sp));
         result += f(from);         result += f(from, ac, mids, sp);
         ++from;                    ++from;
      }                          }
   }                          }

                              static int sum(int from, int to) {
                                 AC ac = getOrCreateAC(Thread.currentThread());
                                 return sum(from, to, ac, new MID[STACKSIZE], 0);
                              }
}                          }
```

Fig. 7. Rewriting example: Program transformations for sampling-based profiling

of executed bytecodes and by minimizing the insertion of conditionals, because such optimizations would reduce the accuracy of the generated sampling profiles.

5.3 Rewriting Example

Fig. 7 illustrates the program transformations performed by Komorium with the same example as in Section 4.4. `STACKSIZE` is a constant that defines the maximum depth of a reified stack; typical values are between 1 000 and 10 000.

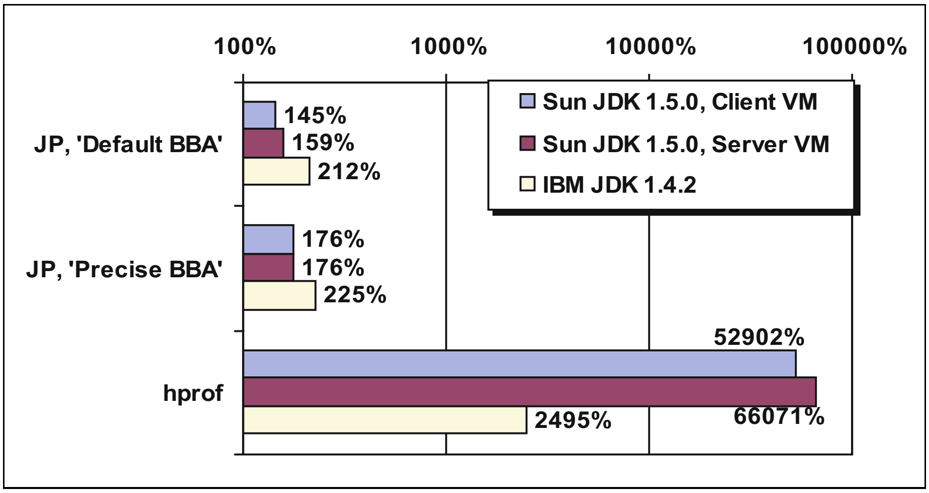


Fig. 8. JP: Profiling overhead for different profiler settings and JDKs

6 Evaluation

In the following we present our measurement results for JP and Komorium. While Section 6.1 provides performance measurements for JP, Section 6.2 discusses the accuracy of sampling profiles as well as the overhead caused by Komorium.

6.1 Exact Profiling (JP)

To evaluate the overhead caused by our exact profiler JP, we ran the SPEC JVM98 benchmark suite [17]. We removed background processes as much as possible in order to obtain reproducible results. For all settings, the entire JVM98 benchmark suite (consisting of several sub-tests) was run 10 times, and the final results were obtained by calculating the geometric mean of the median of each sub-test. Here we present the measurements made with Sun JDK 1.5.0 Client VM, Sun JDK 1.5.0 Server VM, as well as with IBM JDK 1.4.2.

Fig. 8 shows the profiling overhead for two different settings of JP, using the 'Default BBA' resp. the 'Precise BBA'. For both settings of JP, we used a simple profiling agent with the highest possible profiling granularity (the profiling agent was invoked by each thread after the execution of approximately $2^{31} - 1$ bytecodes). Upon program termination, the agent integrated the MCTs of all threads and wrote the resulting profile into a file using a JVM shutdown hook. Depending on the JVM, the average overhead for JP using the 'Default BBA' is 145–212%. For the 'Precise BBA' the average overhead is slightly higher (176–225%). We experienced the highest overhead of 900–1 414% with the 'mtrt' benchmark.

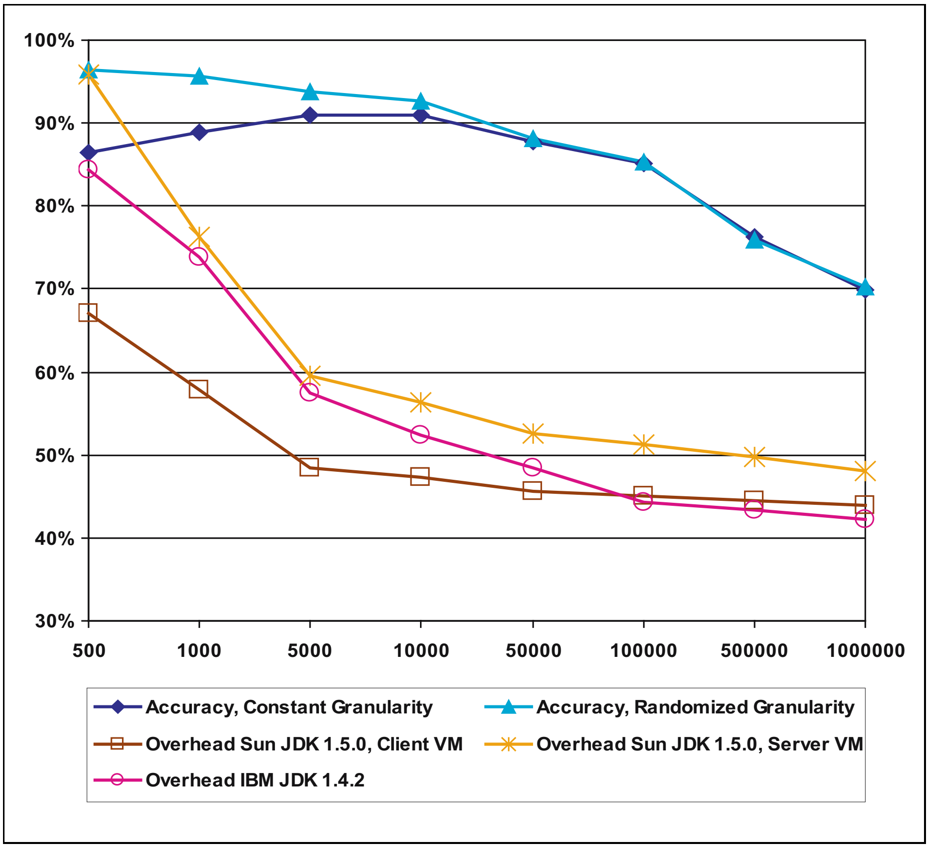


Fig. 9. Komorium: Average profile accuracy (overlap percentage) and average profiling overhead for different profiling granularities (X axis) and JDKs

To compare our profiler with a standard profiler based on the JVMPI/JVMTI, we also evaluated the overhead caused by the 'hprof' profiling agent shipped with the standard JDKs. On Sun's JVMs we started the profiling agent 'hprof' with the '`-agentlib:hprof=cpu=times`' option, which activates JVMTI-based profiling (available since JDK 1.5.0), whereas on IBM's JVM we used the '`-Xrunhprof:cpu=times`' option for JVMPI-based profiling. The argument '`cpu=times`' ensures that the profiling agent tracks every method invocation, as our profiling scheme does.

Because the overhead caused by the 'hprof' profiling agent is 1–2 orders of magnitude higher than the overhead caused by JP, Fig. 8 uses a logarithmic scale. On average, the slowdown due to the 'hprof' profiler is 52 902–66 071% on Sun's JVMs and 2 495% on IBM's JVM. For 'mtrt', the overhead due to 'hprof' exceeds 300 000% on both Sun JVMs.

6.2 Sampling Profiling (Komorium)

Even though the overhead caused by JP is 1–2 orders of magnitude lower than the overhead due to prevailing exact profilers, it may still be too high for certain complex applications. Komorium aims at computing accurate sampling profiles with lower overhead than JP. Fig. 9 shows the average profile accuracy and profiling overhead of Komorium for different profiler settings and JVMs.

We use an *overlap percentage* metric as defined in [3,11] to compare profiles. The overlap represents the percentage of profiled information weighted by execution frequency that exists in both profiles. Two identical profiles have an overlap percentage of 100%. In Fig. 9 we show the average accuracy of sampling profiles (i.e., the overlap percentage of sampling profiles created by Komorium with the corresponding perfect profiles generated by JP using the 'Precise BBA') obtained with different profiling granularities for the SPEC JVM98 benchmark suite.

With a constant profiling granularity, the achievable accuracy is 91%. The best results are obtained with a profiling granularity of 5 000–10 000. As one could expect, the accuracy decreases with increasing profiling granularity (i.e., fewer samples). Surprisingly, also for lower granularities (500–1 000), the accuracy decreases. This is because program behaviour may correlate with our deterministic sampling mechanism, reducing the accuracy of profiles [3].

The accuracy can be improved by adding a small random value r to the profiling granularity [2]. For our measurements, r was equally distributed with $0 \leq r < 100$. This randomization increases the average overlap percentage for lower profiling granularities up to 96%. In order to enable reproducible results despite of the randomization, the profiling agent may use a separate pseudo-random number generator for each thread and initialize it with a constant seed.

For all measured profiling granularities, Komorium causes less overhead than JP. With a profiling granularity of 10 000, Komorium achieves a good tradeoff between high accuracy (an average overlap percentage with perfect profiles of more than 90%) and reasonable overhead of about 47–56% on average (depending on the JVM).

We also evaluated the standard 'hprof' profiler in its sampling mode. On JDK 1.4.2 (JVMPI-based profiling), the average overhead is about 150%, while on JDK 1.5.0 (JVMTI-based profiling) the average overhead is about 90%. These overheads are higher than the overheads caused by Komorium with a profling granularity of 10 000. Moreover, the accuracy of the sampling profiles generated by 'hprof' is very low: On average, the overlap percentage of the sampling profiles with exact ones (generated by 'hprof' in its exact profiling mode) is below 7%. The primary reason for this inferior accuracy is the low sampling rate used by 'hprof' (max. 1 000 samples/second).

7 Limitations and Future Work

While in Section 2 we stressed the benefits of our profiling framework, this section discusses its limitations and outlines some ideas for future improvements.

The major hurdle of our approach is that it cannot directly account for the execution of native code. For programs that heavily depend on native code, the generated profiles may be incomplete. This is an inherent problem of our approach, since it relies on the transformation of Java code and on the counting of the number of executed bytecodes.

Concerning our exact profiler JP, which does not limit the depth of the generated MCTs, memory consumption may be an issue in the case of deep recursions. However, the developer may resort to our sampling profiler Komorium, for which the extra memory needed per thread is constant. Moreover, the size of the generated sampling profiles is factor 2–10 smaller than the size of the corresponding exact profiles, even for the lowest profiling granularity we measured (500).

Finally, we note that bytecode counting and CPU time are distinct metrics for different purposes. While profiles based on bytecode counting are platform-independent, reproducible, directly comparable across different environments, and valuable to gain insight into algorithm complexity, more research is needed in order to assess to which extend and under which conditions these profiles also allow an accurate prediction of actual CPU time for a concrete system.

The value of our profiling tools would further increase if we could use profiles based on bytecode instruction counting to accurately estimate CPU time on a particular target system. This would enable a new way of cross-profiling. The developer could profile an application on his preferred platform $P_{develop}$, providing the profiler some configuration information concerning the intended target platform P_{target}. The profile obtained on $P_{develop}$ would allow the developer to approximate a CPU time-based profile on P_{target}.

For this purpose, individual (sequences of) bytecode instructions may receive different weights according to their complexity. This weighting is specific to a particular execution environment (hardware and JVM) and can be generated by a calibration mechanism. However, the presence of native code, garbage collection, and dynamic compilation may limit the achievable accuracy. Therefore, we will focus first on simple JVM implementations (interpreters), such as JVMs for embedded systems, which do not involve complex optimization and re-compilation phases.

8 Related Work

Fine-grained instrumentation of binary code has been used for profiling in prior work [4,13]. In contrast, all profilers based on a fixed set of events such as the one provided by the JVMPI [15] are restricted to traces at the granularity of the method call. This restriction also exists with the current version of our profilers and is justified by the fact that object-oriented Java programs tend to have shorter methods with simpler internal control flows than code implemented in traditional imperative languages.

The NetBeans Profiler[6] integrates Sun's JFluid profiling technology [9] into the NetBeans IDE. JFluid exploits dynamic bytecode instrumentation and code hotswapping in order to turn profiling on and off dynamically, for the whole application or just a subset of it. However, this tool needs a customized JVM and is therefore only available for a limited set of environments.

[6] `http://profiler.netbeans.org/index.html`

While Komorium is intended as a profiling tool for Java developers, sampling-based profiling is often used for feedback-directed optimizations in dynamic compilers [3,18], because in such systems the profiling overhead has to be reduced to a minimum in order to improve the overall performance. The framework presented in [3] uses code duplication combined with compiler-inserted, counter-based sampling. A second version of the code is introduced which contains all computationally expensive instrumentation. The original code is minimally instrumented to allow control to transfer in and out of the duplicated code in a fine-grained manner, based on instruction counting. This approach achieves high accuracy and low overhead, as most of the time the slightly instrumented code is executed. Implemented directly within the JVM, the instruction counting causes only minimal overhead. In our case, code duplication would not help, because we implement all transformations at the bytecode level for portability reasons. The bytecode instruction counting itself contributes significantly to the overhead.

Much of the know-how worked into JP and Komorium comes from our previous experience gained with the Java Resource Accounting Framework, Second Edition (J-RAF2) [6,12], which also uses bytecode instrumentation in order to gather dynamic information about a running application. J-RAF2 only maintains a single bytecode counter for each thread, comparable to the activation counter used by our profilers (type `AC`). When the number of executed bytecodes exceeds a given threshold, a resource manager is invoked which implements a user-defined accounting or control policy. However, J-RAF2 is not suited for profiling: J-RAF2 cannot compute MCTs, and experiments with a sampling profiler based on J-RAF2 (using the `Throwable` API to obtain stack traces in the user-defined resource manager) were disappointing (low accuracy and high overhead).

9 Conclusion

In this paper we presented a novel profiling framework for Java, consisting of an exact and a sampling profiler. Our profilers exploit the number of executed bytecodes as platform-independent profiling metric. This metric is key to generate reproducible and directly comparable profiles, easing the use of the profiling tools. Moreover, this profiling metric allowed us to implement the profiling framework in pure Java, achieving compatibility with standard JVMs. Our profiling framework supports user-defined profiling agents in order to customize the creation of profiles. In contrast to many existing profiling interfaces which require profiling agents to be written in native code, we support portable profiling agents implemented in pure Java. The activation of the profiling agents follows a deterministic scheme, where the agents themselves control the activation rate in a fine-grained manner.

Performance evaluations revealed that our exact profiler causes 1–2 orders of magnitude less overhead than prevailing exact profilers. Nonetheless, for complex systems the execution time and memory overheads due to our exact profiler may still be too high. For these cases, our sampling profiler is able to generate highly accurate profiles with an overhead lower than standard sampling-based profilers, which often produce inferior profiles.

Acknowledgements

Thanks to Jarle Hulaas who evaluated the overhead caused by the exact profiler JP.

References

1. G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96. ACM Press, 1997.
2. J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, R. Sites, V. Vandervoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4), Nov. 1997.
3. M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.
4. T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.
5. T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.
6. W. Binder and J. Hulaas. A portable CPU-management framework for Java. *IEEE Internet Computing*, 8(5):74–83, Sep./Oct. 2004.
7. W. Binder, J. G. Hulaas, and A. Villazón. Portable resource control in Java. *ACM SIGPLAN Notices*, 36(11):139–155, Nov. 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).
8. J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Proceedings of the ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 21–31. ACM Press, 1999.
9. M. Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance*, pages 139–150. ACM Press, 2004.
10. B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, Nov. 2003.
11. P. Feller. Value profiling for instructions and memory locations. Master Thesis CS1998-581, University of California, Sa Diego, Apr. 1998.
12. J. Hulaas and W. Binder. Program transformations for portable CPU accounting and control in Java. In *Proceedings of PEPM'04 (2004 ACM SIGPLAN Symposium on Partial Evaluation & Program Manipulation)*, pages 169–177, Verona, Italy, August 24–25 2004.
13. J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software–Practice and Experience*, 24(2):197–218, Feb. 1994.
14. S. Liang and D. Viswanathan. Comprehensive profiling support in the Java virtual machine. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS-99)*, pages 229–240, Berkeley, CA, May 3–7 1999. USENIX Association.
15. Sun Microsystems, Inc. Java Virtual Machine Profiler Interface (JVMPI). Web pages at `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/`.
16. Sun Microsystems, Inc. JVM Tool Interface (JVMTI). Web pages at `http://java.`sun.com/j2se/1.5.0/docs/guide/jvmti/.
17. The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Web pages at `http://www.spec.org/osg/jvm98/`.
18. J. Whaley. A portable sampling-based profiler for Java Virtual Machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 78–87. ACM Press, June 2000.

Race Conditions in Message Sequence Charts

Chien-An Chen, Sara Kalvala, and Jane Sinclair

Department of Computer Science,
University of Warwick,
Coventry CV4 7AL, UK
{cssdc, sk, jane}@dcs.warwick.ac.uk

Abstract. Message Sequence Charts (MSCs) are a graphical language for the description of scenarios in terms of message exchanges between communicating components in a distributed environment. The language has been standardised by the ITU and given a formal semantics by means of a process algebra. In this paper, we review a design anomaly, called race condition, in an MSC specification and argue that the current solution correcting race conditions is too weak when implementation is considered. In this paper, we provide an algorithm on partial orders as our solution. The result is a strengthened partial order, which is race-free and remains race-free in the implementation.

1 Introduction

Message Sequence Charts (MSCs) [12] are a trace language, describing scenarios where messages are exchanged between communicating entities in a distributed environment. The language was designed to supplement SDL [18] by providing a graphical representation of behavioural aspects in an SDL specification. The formalism has been recommended as a standard by the ITU (International Telecommunication Union). During the last decade, it has evolved incrementally from a plain message exchange diagram to a multi-layered documentation methodology with rich constructs. Due to their readability and tool support, MSCs have become a specification language in their own right and have been enjoying a widespread use in specifying telecommunication protocols and reactive systems, particularly system requirements during early stages of development.

In addition to their industrial popularity, MSCs have been drawing much attention from researchers. Attempts at a formal semantics of MSCs have emerged since MSC'92 [6]. Various approaches have been adopted for this task, such as automata theory [13,14], Petri Nets [10], streams [5] and process algebra [7,9,15]. Synthesising system models or behavioural models from MSC scenarios is also an active topic [1,19,20]. MSC specifications can be syntactically analysed for a variety of design anomalies, such as deadlocks, race conditions [2], process divergence and non-local branching choices [3].

In this paper, we are concerned only with race conditions and their solutions. The semantics of an MSC used here, called causal ordering, is actually a partial order characterising execution traces on communication events. A race condition refers to an inconsistency between the causal ordering specified in an MSC and the ordering that may occur in practice. The current solution [16] to correcting race conditions asserts that given a causal ordering there exists a unique race-free partial order that is a minimal weakening of the causal ordering. Dually, there also exists a unique race-free partial order that is a minimal strengthening of the causal ordering. Both partial orders have syntactically legal MSCs to match.

Here we focus on the strengthening counterpart of a causal ordering. We have observed that if enforcing the strengthened ordering properties by adding extra messages as acknowledgements, new race conditions may occur. In this situation, more acknowledgement messages are required, so that a simple MSC diagram can be flooded with unnecessary messages and becomes hard to read and analyse. Such an observation motivates our work. In this paper, we propose an algorithm on partial orders as the solution. In addition, justification on the algorithm is also provided before we implement the approach for tool-support. The result of our work shows that for a causal ordering, there exists a minimally strengthened partial order, that is race-free and remains race-free when acknowledgement messages are added to enforce the strengthened ordering properties. Our approach enables specifiers to illustrate explicitly more design details in an MSC scenario without the risk of race conditions. Although we use MSCs as the central language in this paper, our approach can be applied to other languages with partial-order semantics, e.g. UML Sequence Diagrams [4] and LSCs [8].

This paper is organised as follows. Section 2 gives an overview of the MSC language we use in this paper and its semantics. The concept of race conditions and the current solutions are also introduced. In section 3, we present the remaining problems which motivate our work; in section 4, we propose an approach to correcting race conditions without introducing new ones when acknowledgement messages are added. Conclusions are presented in section 5. Familiarity with the theories of relations and partial orders is presumed.

2 Preliminary

2.1 Message Sequence Charts

A *basic* MSC (bMSC) is a building block for an MSC document. As can be seen in Fig.1(a), $MSC1$ contains three *instances* (or *processes*), namely $P1$, $P2$ and $P3$, denoted by vertical axes. Three *messages*, a, b and c, denoted by arrows, are exchanged between those instances. The frame around the diagram represents the *environment*. $MSC1$ intuitively describes a scenario where $P1$ sends message a to $P2$; after receiving, $P2$ sends messages b and c to $P3$ and $P1$ respectively.

A temporal ordering is defined along each vertical axis and horizontal arrow in the sense that (1) the events along an instance axis proceed from top to bottom, and (2) a message must be sent before it is received. In the bMSC convention, it is assumed that the communication medium between distributed processes is reliable, i.e. no message gets lost. Furthermore, the setting of a bMSC is assumed to be of *asynchronous* communication. Hence in $MSC1$ of Fig.1(a), the order of receiving b and c is undefined. The ordering properties can be strengthened by a *general ordering* construct, denoted by a dashed line with an arrowhead in the middle. A general ordering symbol is attached to the events that need to be ordered. As illustrated in $MSC2$ of Fig.1(b), the general ordering adds a constraint that receiving c has to occur after receiving b.

The core subset of bMSCs we use is described in the ITU-standard [12]. Our definition is in a similar style to those that can be found in the early MSC-related work, e.g. [1,2]. The bMSC notation used in this paper is defined as follows.

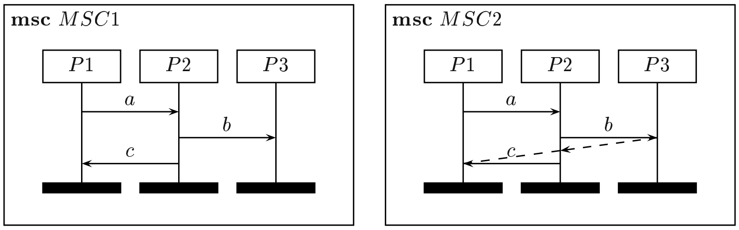


Fig. 1. (a)A bMSC diagram (b)A bMSC with a general ordering

Definition 1. *A bMSC is defined as a tuple $\langle \mathcal{P}, \mathcal{M}, msg, \mathcal{O}, <_C \rangle$ where:*

- *$\mathcal{P}$ is a finite set of instances, i.e. $\mathcal{P} = \{Pi \mid i \in 1..n\}$.*
- *$\mathcal{M}$ is a finite set of message names. A set Σ for all the events in a bMSC can be derived accordingly as $\Sigma = \Sigma_{out} \cup \Sigma_{in}$ where $\Sigma_{out} = \{!a_{ij} \mid i, j \in 1..n \wedge a \in \mathcal{M}\}$ and $\Sigma_{in} = \{?a_{ij} \mid i, j \in 1..n \wedge a \in \mathcal{M}\}$. The label $!a_{ij}$ denotes an output event that the instance Pi sends a message 'a' to Pj, and similarly an input event $?a_{ij}$ means Pj receives a message 'a' from Pi.*
- *$msg : \Sigma_{out} \rightarrow \Sigma_{in}$ is a bijection matching each output event to its corresponding input event, i.e. $msg = \{x \in \mathcal{M}, i, j \in 1..n \bullet !x_{ij} \mapsto ?x_{ij}\}$.*
- *$\mathcal{O} : \mathbb{P}(\Sigma \times \Sigma)$ is a relation on Σ, recording the constraint of general orderings in a bMSC. For a pair $x \mapsto y : \Sigma \times \Sigma$, $x \mapsto y \in \mathcal{O}$ if and only if there exists a general ordering symbol from x to y.*
- *$<_C$: $\mathbb{P}(\Sigma \times \Sigma)$ is a partial order on Σ. For each $Pi \in \mathcal{P}$, an adjacency relation $<_i$ denotes the top-to-bottom temporal ordering of the events occurring on Pi. The partial order $<_C$ is the transitive closure of the relation $\ll_C$ defined as*

$$\ll_C = \bigcup_{i:1..n} <_i \cup\, msg \cup \mathcal{O}.$$

The partial order $<_C$ is also known as the *causal ordering*, which can be understood as a visual order displayed in a bMSC diagram. The semantics of a bMSC specifically refers to its causal ordering. Unless specified otherwise, the partial orders in this paper are *strict* in the sense that they are *anti-reflexive.* Also note that the subscripts of the event labels, recording the origin and destination of a message, are of no importance here and therefore can be ignored. For a message $x \in \mathcal{M}$, its input and output events are represented as $?x$ and $!x$ respectively.

A collection of bMSCs can be composed together sequentially or conditionally. The High-level MSC, also known as hMSC or *road map*, is a structuring mechanism to compose bMSCs. In this paper, we only consider bMSCs, and the term MSC specifically refers to a bMSC.

2.2 Race Conditions

In discussions of operating systems or distributed systems, where at least two parallel processes have access to a single resource simultaneously, the term *race condition* refers to the situation where, without a synchronisation mechanism, inconsistencies may arise depending on which process wins the race to communicate with the resource. In the context of MSCs, however, a race condition does not match its usual meaning and therefore needs further explanation.

Initially discussed in [2], a race condition in an MSC refers to the likelihood that the implementation fails to obey the causal ordering described in its MSC specification. The concept can be illustrated via an example. $MSC3$ in Fig.2(a) shows a scenario with a race condition between the events $!b$ and $?c$. The specification requires that the input of c must follow the output of b. Nevertheless, $P3$ is specified to send out c after receiving a. Without querying $P2$, $P3$ has no knowledge of when $P2$ sends b. Therefore, in the implementation of such a protocol, it is quite possible that the message c arrives at $P2$ before $P2$ starts to send out b, which contradicts the specification.

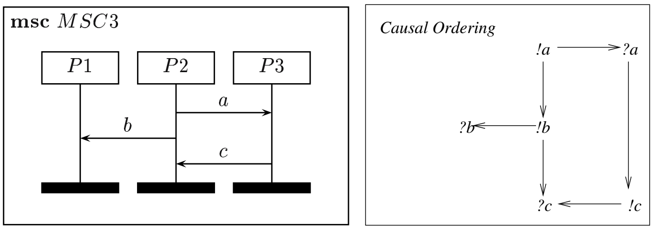


Fig. 2. (a)MSC with a race condition (b)Causal ordering $<_C$ of $MSC3$

Formal descriptions of a race condition can be found in the work of Alur et al. [2] and Mitchell [16] in different styles. In addition to causal ordering, Alur et al. [2] have defined two other levels of observation, i.e. *inferred ordering* and *enforced ordering*. In their method, detection of race conditions consists in checking if the inferred ordering is a subset of the transitive closure of the enforced ordering. They have also proved that detection of race conditions in a basic MSC[1] is decidable, and the tool uBET [11] from Bell Lab has been developed accordingly to address this problem.

Yet a solution which attempts to correct race conditions in an MSC had not emerged until Mitchell's work [16]. Here we quote directly the definition

[1] See [17] for detection of race conditions in High-level MSCs.

of a race condition in [16]. Intuitively, an MSC is race-free iff for an event x occurring before an input event $?e$, x must precede its corresponding output event $!e$ provided that x is not $!e$.

Definition 2. *An MSC is race-free when its causal ordering $<_C$ is race-free. A partial order $<$ on Σ is race-free if and only if*

$$x <?e \Rightarrow (x <!e \vee x =!e)$$

for every event $x \in \Sigma$ and message $e \in \mathcal{M}$.

In Mitchell's work [16], two solutions are proposed when given a causal ordering with race conditions. One is based on a race-free *inherent causal ordering* $<_I$, which is a minimal weakening of the causal ordering $<_C$. The other is the existence of a race-free *inherent refinement ordering* $<_R$, which is a minimal strengthening of $<_C$. The overall relationship is $<_I \subseteq <_C \subseteq <_R$. Both $<_I$ and $<_R$ have syntactically legal MSCs to match. The mapping between an ordering and its MSC format is trivial and can be automated. Here we are only concerned with $<_R$. The following definition differs from that in [16] to maintain $<_R$ a unique minimal strengthening of a causal ordering.[2]

Definition 3. *For a causal ordering $<_C$ on Σ, its inherent refinement ordering $<_R$ is the transitive closure of the relation $\ll_R$ defined as*

$$\ll_R = \ll_C \cup \{(x, !e) \in \Sigma \times \Sigma_{out} \mid x \ll_C \ ?e \text{ and } \neg\ (x \leq_C \ !e)\}.$$

Notations on partial orders. For a causal ordering $<_C$ on Σ and $x <_C y$ where $x, y \in \Sigma$, we say that x is an *immediate predecessor* of y if and only if $x \ll_C y$. For illustration, however, we use directed graphs to depict the relevant partial orders. Each vertex represents an element in Σ, and a directed edge is drawn from x to y, denoted by $x \rightarrow y$, whenever $x \ll_C y$. A path exists from x to y, denoted by $x \rightsquigarrow y$, iff $x <_C y$. Since $<_C$ is anti-reflexive, we have $x \not\rightsquigarrow x$, meaning a path consists of at least one edge in the graph. We let $\leq_C = <_C \cup Id(\Sigma)$ denote the reflexive version of $<_C$ such that $x \leq_C x$. The symbol $\rightsquigarrow_0$ is used to denote a path with zero or more edges, i.e. $x \rightsquigarrow_0 y$ iff $x \leq_C y$. The graph we use here is a variant of the *Hasse diagram* for a partial order in the sense that (1) the shape is different in order to maintain the similarity between a partial order and its MSC format, and (2) the edges are arrow-headed so that two events are ordered iff there is a path between them. For example, Fig.2(b) is the graph for the causal ordering $<_C$ of *MSC*3. It can be observed that the mapping between the directed graph and its MSC format is trivial. Also note that the variation pattern between $<_C$, $\ll_C$ and $\leq_C$ also applies to $<_R$.

The intuitive idea of constructing $<_R$ from $<_C$ of an MSC is to add ordering properties into $<_C$ so that the resulting $<_R$ satisfies Definition 2. Note that we

[2] Soundness problems of the original definition for the inherent refinement ordering in [16] will be discussed in our future exposition.

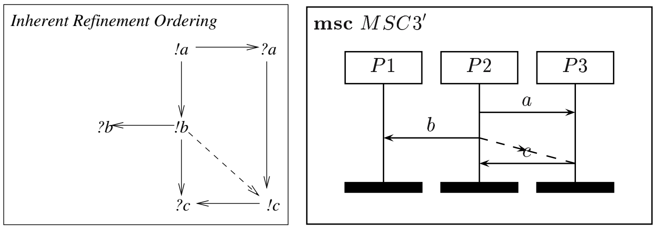


Fig. 3. (a)Inherent refinement ordering $<_R$ of *MSC*3 (b)Matching MSC scenario

use dashed edges for these additional constraints in a graph. Since the added ordering properties are facilitated by general ordering symbols in MSCs, dashed edges can maintain the resemblance between an inherent refinement ordering and its MSC format. As seen in Fig.2(b), $<_C$ of *MSC*3 has a race condition since $!b \rightsquigarrow ?c$ but $!b \not\rightsquigarrow !c$. The solution $<_R$ is constructed in Fig.3(a) by adding a dashed edge between $!b$ and $!c$, so that $<_R$ is race-free because $!b \rightsquigarrow ?c$ and $!b \rightsquigarrow !c$. The MSC scenario matching $<_R$ is *MSC*3′ in Fig.3(b).

3 Motivation

In the previous section, we have demonstrated that given an MSC scenario *MSC*3, its race-free counterpart *MSC*3′ can be derived by constructing $<_R$. A general ordering is used to delay events to avoid a race condition. In this case, $!c$ is delayed until $!b$ occurrs, so that *MSC*3′ is race-free. Nevertheless, further problems may arise when we consider how a general ordering can be implemented in a distributed environment. Mitchell [16] has indicated that the choice of implementation is up to the system designers who may use any mechanism that they deem *appropriate* for a particular circumstance. This effect can always be achieved by adding extra messages into the MSC with general orderings.

We recall the basic assumption of the MSC setting. Instances communicate asynchronously in a distributed environment without sharable resources. An instance can only get the information from others via message passing. Therefore, the above approach of adding messages is an effective way for specifiers to reveal explicitly how to implement the general orderings in an MSC specification. For example, in order to enforce the general ordering $!b \mapsto !c \in \mathcal{O}$ in *MSC*3′, an acknowledgement message can be added as shown in *MSC*3″ of Fig.4(a), where a silent symbol τ is used to label such a message since it does nothing but maintain the ordering. It can be easily noted that *MSC*3″ is not race-free since $?a <_C ?\tau_1$ but $?a \not<_C !\tau_1$, where $<_C$ is the causal ordering of *MSC*3″.

The problem continues even if we keep on building the inherent refinement ordering of *MSC*3″, which adds a general ordering between $?a$ and $!\tau_1$. Enforcing the general ordering with another message, say τ_2, gives us *MSC*3‴ as in Fig.4(b), where a race condition still exists between $!b$ and $?\tau_2$. It goes back to the case of *MSC*3. Without a more sophisticated approach, a simple MSC scenario can be flooded with unnecessary messages and becomes unreadable.

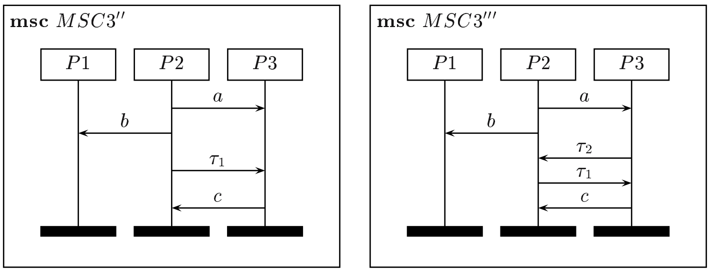


Fig. 4. (a)An acknowledgement message (b)MSC flooded with messages

In brief, our work is motivated by the observation that for a race-free inherent refinement ordering of an MSC with race conditions, an implementation that enforces the general orderings may not be race-free. The following definition explains the term *implementation* we use hereafter under the assumption that there is no general ordering symbols in the original MSC. Notationally, we use $Rel\, S$ as shorthand for the set of relations on S, i.e. $Rel\, S = \mathbb{P}(S \times S)$. We define a set A consisting of the events caused by the acknowledgement messages, i.e. $A = \{!\tau_i \mid i \in \mathbb{N}\} \cup \{?\tau_j \mid j \in \mathbb{N}\}$. The symbol $|S|$ denotes the number of elements in set S. For a relation R on set S, R^* is the transitive closure of R.

Definition 4. *For a causal ordering $<_C$ where $\mathcal{O} = \emptyset$ and its inherent refinement ordering $<_R$, the implementation of $<_R$ is a function $IMP : Rel\Sigma \rightarrow Rel(\Sigma \cup A)$ such that*

$$IMP(<_R) = (\Upsilon_1(\ll_R))^*$$

where the function $\Upsilon : Rel\Sigma \rightarrow Rel(\Sigma \cup A)$ is defined as follows. For a relation R on Σ, a message $x \in \mathcal{M}$, events $v_1, v_2 \in \Sigma$ and a counter $i \in 1..(|\ll_R \backslash \ll_C| + 1)$

$$\Upsilon_i(\ll_C) = \ll_C$$
$$\Upsilon_i(\{v_1 \mapsto !x\} \cup R) = \Upsilon_{i+1}(R) \cup \{!\tau_i \mapsto ?\tau_i, v_1 \mapsto !\tau_i, ?\tau_i \mapsto !x, v_2 \mapsto ?\tau_i\}$$

where $v_1 \mapsto !x \in \ll_R \backslash \ll_C$, $v_1 \mapsto !x \notin R$ and $v_2 \mapsto !x \in \ll_C$.

The intuition behind the above definition is to construct a causal ordering of the MSC in which an acknowledgement message is used to enforce a general ordering symbol. Here we use two graphs as an example. The relation $<_R$ depicted in Fig.5(a) shows the inherent refinement ordering of $MSC3$. A dashed edge denotes the difference between $\ll_C$ and $\ll_R$. The partial order $IMP(<_R)$ is displayed in Fig.5(b) by replacing the dashed edge $!b \mapsto !c$ in $<_R$ with $!\tau_1 \mapsto ?\tau_1$. Note that $IMP(<_R)$ is not race-free since $?a \rightsquigarrow ?\tau_1$ but $?a \not\rightsquigarrow !\tau_1$. The following proposition formalises this observation that motivates our work.

Proposition 1. $\neg$ ($\forall <_R$: *$Rel\Sigma$, $<_R$ is race-free $\Rightarrow$ $IMP(<_R)$ is race-free*).

This proposition can be easily justified by the counter-example we show in Fig.5. In the next section, we propose an approach to finding a partial order $<_{RF}$ that is stronger than $<_R$ such that both $<_{RF}$ and $IMP(<_{RF})$ are race-free.

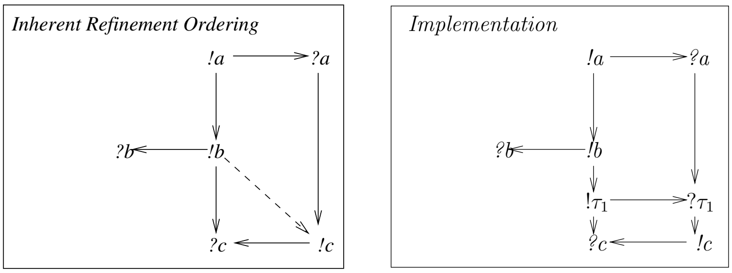


Fig. 5. (a)$<_R$ of *MSC*3 (b)*IMP* of $<_R$ of *MSC*3

4 Correcting Race Conditions

In this section, we propose our solution for correcting race conditions in an MSC scenario. The algorithm is expressed in a functional style with the aid of graphs explaining how the algorithm works. Soundness is then discussed in a more abstract way. Our intention is to make a formal argument to establish the correctness of our approach before coding it. Finally, a couple of simple examples are given for illustration.

4.1 Race-Free Refinement

We have observed that although an inherent refinement ordering is race-free, its implementation may not be. Our goal is to find another strengthened partial order $<_{RF}$, called *race-free refinement*, such that $<_R \subseteq <_{RF}$ and both $<_{RF}$ and $IMP(<_{RF})$ are race-free. We achieve this task by defining a function RF, which takes an inherent refinement ordering and non-deterministically returns a race-free refinement, i.e. $<_{RF} = RF(<_R)$. For simplicity, our approach is under the assumption that there is no general ordering construct in the original MSC diagram, which means all dashed edges appearing in the graphs are added by inherent refinement orderings.

The basic concept behind our approach can be understood via the two graphs in Fig.6. The symbol v stands for an event that can be either input or output. In Fig.6(a), the triangle $(v, !x, ?x)$ is a building block of all the inherent refinement orderings. The implementation of this graph will add a new input event, say $?\tau$, between $!y$ and $!x$. This addition causes a risk of a new race condition because $?\tau$ is a successor of $!y$, but $!y$ may not precede its corresponding output event $!\tau$, which is a successor of v. Nevertheless, no new race condition will occur if we have $?\tau$ precede $!y$, which means the dashed arrow should be lifted up to link v and $!y$ as shown in Fig.6(b). The new graph is still race-free because $v \rightarrow !y$ and $!y \rightarrow !x$ implies $v \rightsquigarrow !x$. This trivial example shows the most basic operation of the function RF, i.e. lifting up the dashed arrows in $<_R$ to a preceding output event. In a life-size MSC, however, there may be many output events preceding $!x$. In this case, we pick up the earliest one which does not precede v. To make this procedure more precise, we define the function ρ to perform this task.

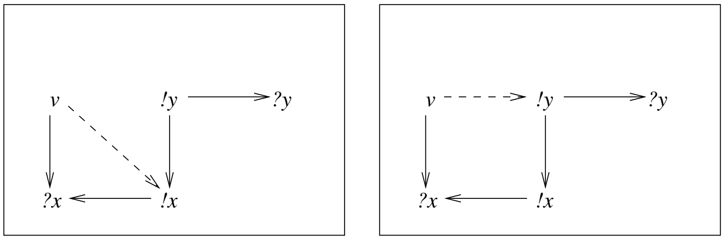


Fig. 6. (a)Graph for $<_R$ (b)Graph for $RF(<_R)$

Definition 5. *For a partial order $<$ on Σ and two events $u_1, u_2 \in \Sigma$, the function $\rho : Rel\Sigma \times \Sigma \times \Sigma \to \mathbb{P}(\Sigma_{out})$ is defined*

$$\rho(<, u_1, u_2) = min(\{v : \Sigma_{out} \mid v \leq u_2 \wedge v \not\leq u_1\})$$

where $min(S)$ returns a set of minimal elements of a partially ordered set S in the sense that an element a in S is called a minimal element if no other element of S strictly precedes a.

The function renders a set of minimal output events such that the events precede u_2 but do not precede or equal to u_1 with respect to the partial order $<$. The second parameter u_1 stands for the pivot-like event as v in Fig.6. The third parameter u_2 holds a place for the starting event, like $!x$, from which we trace back to a preceding output event. The application of ρ on the above example gives us $\rho(<_R, v, !x) = \{!y\}$ where $<_R$ represents the partial order depicted in Fig.6(a). Nevertheless, this example is too trivial in the sense that there is no event preceding $!y$. Simply replacing $v \mapsto !x$ with $v \mapsto !y$ gives us a race-free refinement. If we consider the case where some events precede $!y$, we need another mechanism to check whether the newly added edge $v \mapsto !y$ will cause a new race condition or not in the implementation. This mechanism can be explained more clearly after the function RF is defined.

Conventions on notation and designation are as follows. We let R, S range over $Rel\Sigma$. The lower-case letters e, r range over $\Sigma \times \Sigma$ and u, v over Σ. For a pair $e \in R$, $e.s \in \Sigma$ stands for the source vertex of the edge e and $e.d \in \Sigma$ for the destination vertex. We also define a replacement operator $[/]$ on a set S in the sense that $S[x/y] = \{x\} \cup S\backslash\{y\}$.

Definition 6. *For a causal ordering $<_C$ where $\mathcal{O} = \emptyset$ and its corresponding inherent refinement ordering $<_R$, the function $RF : Rel\Sigma \to Rel\Sigma$, which maps a partial order to a relation, is defined as*

$$RF(<_R) = (\Phi(\ll_R, \ll_R\backslash\ll_C))^*.$$

The function $\Phi : Rel\Sigma \times Rel\Sigma \to Rel\Sigma$ is defined inductively as

$$\Phi(R, \emptyset) = R$$

$$\Phi(R, \{e\} \cup S) = \begin{cases} \Phi(R,S)\backslash\{e\} & \text{if } e \in (\Phi(R,S)\backslash\{e\})^* \\ \Gamma(\Phi(R,S), r)[r/e] & \text{otherwise} \end{cases}$$

where $e \notin S$ *and*

$$r = e.s \mapsto v \text{ such that } v \in \rho((\Phi(R,S))^*, e.s, e.d).$$

Here we classify the right arrow $\rightarrow$ *into dashed or solid ones. A solid arrow, denoted by* $x \rightharpoonup y$*, describes the ordering formed by message arrows and instance axes. All other arrows are dashed, represented as* $x \rightharpoondown y$*, e.g. the edges in the set* $\ll_R \backslash \ll_C$ *or those generated by the functions* Φ *and* Γ*. Formally,* $\rightharpoonup = \ll_C$ *and* $\rightharpoondown = \rightarrow \backslash \rightharpoonup$ *where* $\rightarrow$ *refers to the binding occurrence of the relation* R *in* Γ*.*

The function $\Gamma : Rel\Sigma \times (\Sigma \times \Sigma) \rightarrow Rel\Sigma$*, which solves new race conditions that may arise when* r *is added by* Φ*, is defined as follows.*

$$\Gamma(R,r) = \begin{cases} R\backslash\{x \mapsto r.d\} & \text{if } \exists\, x : \Sigma_{out}\backslash\{r.s\}, x \rightharpoondown r.d & (1)\\ r_1 \cup \Gamma(R, r_1) & \text{if } \exists\, x : \Sigma_{in}\backslash\{r.s\}, x \rightharpoonup r.d & (2)\\ \Gamma(R, r_1)[r_1/x \mapsto r.d] & \text{if } \exists\, x : \Sigma_{in}\backslash\{r.s\}, x \rightharpoondown r.d & (3)\\ r_2 \cup \Gamma(\Gamma(R, r_1)[r_1/x \mapsto r.d], r_2) & \text{if } \exists\, x, y : \Sigma_{in}\backslash\{r.s\}, & \\ & \quad x \rightharpoondown r.d \wedge y \rightharpoonup r.d & (4)\\ r_1 \cup \Gamma(R, r_1)\backslash\{y \mapsto r.d\} & \text{if } \exists\, x : \Sigma_{in}\backslash\{r.s\}, y : \Sigma_{out}\backslash\{r.s\}, & \\ & \quad x \rightharpoonup r.d \wedge y \rightharpoondown r.d & (5)\\ R & \text{otherwise} & (6) \end{cases}$$

where

$$r_1 = x \mapsto v_1 \text{ such that } v_1 \in \rho(R^*, x, r.s)$$
$$r_2 = y \mapsto v_2 \text{ such that } v_2 \in \rho((\Gamma(R, r_1)[r_1/x \mapsto r.d])^*, y, r.s).$$

The algorithm for finding a race-free refinement consists of two recursive function calls, i.e. Φ and Γ. For each dashed edge in $<_R$, the basic operation of lifting that edge to a preceding output event up to its minimum is performed. This task is achieved by the iteration of the function Φ and can be illustrated in

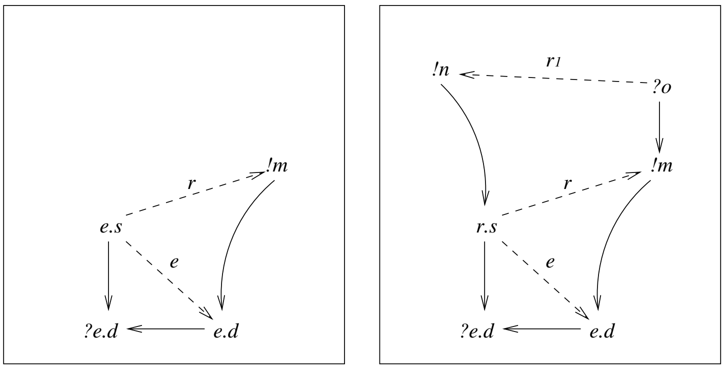


Fig. 7. (a)Behaviour of Φ (b)Behaviour of Γ

a more abstract way as shown in Fig.7(a). The curved edge from $!m$ to $e.d$ means $!m \rightsquigarrow_0 e.d$. The event $!m$ is the minimal element with the constraints such that $!m \rightsquigarrow_0 e.d$ and $!m \not\rightsquigarrow_0 e.s$. In the next section, we will prove in Lemma 1 that given a pair $e \in \ll_R \setminus \ll_C$, such an $!m$ always exists and could be simply $e.d$. The graph in Fig.7(a) depicts the behaviour of Φ, which replaces e with another pair $r = e.s \mapsto !m$. We can observe that an acknowledgement message enforcing r will add an input event preceding $!m$, causing no new race conditions.

The above description shows an ideal scenario where no event immediately precedes $!m$. Nevertheless, in some MSC scenarios, it is not unusual that there is another event immediately preceding $!m$, causing risks of new race conditions in the implementation. More precisely, if an input event, say $?o$, immediately precedes $!m$, the addition of r will *always* cause a new race condition due to $?o \not\rightsquigarrow r.s$, which we will prove later in Proposition 2. On the other hand, if an output event, say $!o$, immediately precedes $!m$, the addition of r will *never* cause a new race condition. We justify this feature here. From the definition of ρ, we can deduce $!o \rightsquigarrow_0 r.s$, otherwise $!o$ will be the $!m$. The acknowledgement message τ adds a pair $!\tau \mapsto ?\tau$ such that $r.s \rightarrow !\tau$ and $!o \rightarrow ?\tau \rightarrow !m$, which is still race-free since $!o \rightarrow ?\tau$ and $!o \rightsquigarrow_0 r.s \rightarrow !\tau$.

In Fig.7(b), we demonstrate the situation where an input event $?o$ immediately precedes $!m$. In this case, a new race condition will always occur in the implementation as mentioned earlier. This feature justifies the existence of the iteration of Γ within each recursive call of Φ. Tackling further race conditions caused by r is the task of Γ depending on how $!m$ is preceded. There are in total six cases expressed in the six equations in Γ. The graph of Fig.7(b) illustrates the equation (2) from Definition 6. The way we solve the problem is by finding another minimal output event, say $!n$, such that $!n \rightsquigarrow_0 r.s$ and $!n \not\rightsquigarrow_0 ?o$, and adding an edge $r_1 = ?o \mapsto !n$. Similarly, we can also prove that there always exists such an $!n$, which could be simply $r.s$. Note that $\Gamma(R, r_1)$ again appears at the right hand side of $=$ in the equation (2), which means checking if any new race condition may occur when r_1 is added. In this case, a new race condition occurs iff there exists at least one input event immediately preceding $!n$. The recursive structure of Γ basically forms an iteration to add or replace a set of edges $r_{1..n}$ after r is added under one recursive call of Φ. A more complicated structure of Γ is the equation (4), where there are two input events immediately preceding $!m$. The approach is similar to the above case except that the iteration of Γ occurs in one recursive call of Γ itself.

4.2 Soundness

Here we provide justification for the function *RF* that we claim can correct race conditions without introducing new ones when extra messages are added to enforce general orderings. Although *RF* is defined on partial orders, the style of our soundness discussion is based on analysing the behaviour of the function on the corresponding graphs, to which two template diagrams in Fig.7 serve as visual aids. The first proposition justifies the existence of Γ in the sense that a new race condition will always occur in implementation if an input event immediately precedes the output event found by ρ in Φ, e.g. $?o \rightarrow !m$ in Fig.7(b).

Proposition 2. *In the context of Γ in RF, if there exists an event $x : \Sigma_{in} \backslash \{r.s\}$ such that $x \rightarrow r.d$, then $x \not\rightsquigarrow r.s$.*

Proof. There are two cases for $x \rightarrow r.d$. One is $x \rightharpoonup r.d$, and the other is $x \rightharpoondown r.d$. We show that $x \not\rightsquigarrow r.s$ in both cases.

Case 1: $x \rightharpoonup r.d$. Assume $x \rightsquigarrow r.s$; there are two possible routes, i.e. $x \rightharpoonup\rightsquigarrow_0 r.s$ and $x \rightharpoondown\rightsquigarrow_0 r.s$. The former case leads to a contradiction since $x \rightharpoonup r.d$ but $r.d \not\rightsquigarrow_0 r.s$. The latter case implies an event $v \in \Sigma$ such that $x \rightharpoondown v \rightsquigarrow_0 r.s$. Due to the nature of $\ll_R$, v must be an output event. So we have $x \rightharpoondown !y$ for an output event $!y$. The structure of $\ll_R$ enable us to deduce $x \rightharpoonup ?y$ from $x \rightharpoondown !y$, which contradicts the fact that $r.d$ is an output event.

Case 2: $x \rightharpoondown r.d$. This case implies $x \rightharpoonup ?r.d$, where $?r.d$ denotes the input event of $r.d$. Assuming $x \rightsquigarrow r.s$, there are two routes, i.e. $x \rightharpoondown\rightsquigarrow_0 r.s$ and $x \rightharpoonup\rightsquigarrow_0 r.s$. The former leads to a contradiction since $x \rightharpoondown r.d$ but $r.d \not\rightsquigarrow_0 r.s$. The latter also results in a contradiction because $x \rightharpoonup\rightsquigarrow_0 r.s$ implies $?r.d \rightsquigarrow_0 r.s$. Since $r.d \rightarrow ?r.d$ and $?r.d \rightsquigarrow_0 r.s$, we get $r.d \rightsquigarrow r.s$, which violates ρ. □

When explaining the mechanism of Φ, we have already mentioned that given a pair $e \in \ll_R \backslash \ll_C$, we can always find a minimal output event v such that $v \rightsquigarrow_0 e.d$ and $v \not\rightsquigarrow_0 e.s$. The same applies to Γ. Referring to Fig.7, we can say that $!m$ always exists in the application of Φ, and so does $!n$ in Γ. In the lemma below, we prove this property by showing that $!m$ could be simply $e.d$, and $!n$ could be simply either $r.s$ or the corresponding output event of $r.s$.

Lemma 1. *In the context of RF, the application of ρ gives a non-empty set.*

Proof. There are two places where the function ρ is called, i.e. Φ and Γ. In the case of Γ, the third parameter $r.s$ can be either an input or an output event.

Case 1: Φ. The event $e.d$ is a legitimate $!m$ because $e.d$ is an output event, and we also have $e.d \not\rightsquigarrow_0 e.s$ and $e.d \rightsquigarrow_0 e.d$.

Case 2: Γ. There are two sub-cases here.

Case 2.1: If $r.s$ is an output event, $r.s$ is a legitimate $!n$. We prove by contradiction. We assume $r.s$ is not a legitimate $!n$, which means $r.s \rightsquigarrow_0 ?o$. Since $?o \rightarrow !m$ and $!m \rightsquigarrow_0 e.d$, we can deduce $r.s \rightsquigarrow e.d$. In this case, the edge e would not have not existed in $<_R$ to correct a race condition in the first place, which is a contradiction.

Case 2.2: If $r.s$ is an input event, the corresponding output event of $r.s$, denoted by $!r.s$ is a legitimate $!n$. Similarly, we assume $!r.s$ is not a legitimate $!n$, i.e. $!r.s \rightsquigarrow_0 ?o$. We know $!r.s \neq ?o$, so $!r.s \rightsquigarrow ?o$. Since $<_R$ is race-free, $!r.s \rightsquigarrow ?o$ implies $!r.s \rightsquigarrow !o$. We also know that $!o \rightsquigarrow_0 r.s$, otherwise $!o$ will be the $!m$. Since $!o \neq r.s$, we have $!o \rightsquigarrow r.s$ hence $!o \rightsquigarrow !r.s$. A cycle arises in $<_R$ due to $!r.s \rightsquigarrow !o$ and $!o \rightsquigarrow !r.s$, which is a contradiction. □

We can observe that the mechanism of RF is to add or replace edges into the graph of an inherent refinement ordering $<_R$. The following lemma asserts that the edges added by RF are finite, which implies that the algorithm terminates. We prove the lemma by showing that each edge that may be added by one recursive call of Γ is between a pair of unordered input and output events.

Lemma 2. *For an inherent refinement ordering $<_R$, the application of RF inserts a finite number of edges into the graph of $<_R$.*

Proof. The structure of RF requires that for an edge e in $\ll_R \backslash \ll_C$, the function Φ replaces e with the edge r, and for each r, a set of edges $r_{1..n}$ may be added by Γ. Since $\ll_R \backslash \ll_C$ is finite, the number of edges replaced by Φ is finite. For an $r_i \in r_{1..n}$ added by Γ, there must exist an r_{i-1} and an input event, say $?o$, such that $?o \to r_{i-1}.d$ and $?o = r_i.s$. Proposition 2 asserts $?o \not\rightsquigarrow r_{i-1}.s$. Since $r_i.d \rightsquigarrow_0 r_{i-1}.s$, we have $?o \not\rightsquigarrow r_i.d$ before r_i is inserted. We also know that $r_i.d \not\rightsquigarrow_0 ?o$ from the definition of ρ. Since $r_i.d \neq ?o$, we have $r_i.d \not\rightsquigarrow ?o$. So an important feature of r_i arises in the sense that r_i only links a pair of unordered input and output events in $<_R$. We let the set of such pairs be U. We can assert $r_{1..n} \subseteq U \subseteq \Sigma \times \Sigma$. Since $\Sigma \times \Sigma$ is finite, the set $r_{1..n}$ is therefore finite. □

With the above lemma, we know that the task of RF consists in adding a finite number of edges into the graph of $<_R$. Here arises another question: how can we ensure that the relation $RF(<_R)$ is still a partial order? For an anti-reflexive relation, its transitive closure is a partial order if and only if there exists no cycle in the graph of that relation. Therefore our next observation is that the relation $RF(<_R)$ is a partial order by proving the graph of $RF(<_R)$ is acyclic in the following proposition.

Proposition 3. *For an $<_R$, the graph of $RF(<_R)$ is acyclic.*

Proof. Since the function RF recursively adds a finite number of edges, $r_{1..n}$, into the adjacency version of an inherent refinement ordering (Lemma 2), we can represent $RF(<_R)$ as $\ll_R \cup r_{1..n}$. We prove by induction on every $n \in \mathbb{N}$ that $RF(<_R) = \ll_R \cup r_{1..n}$ is acyclic.

Base case: Setting $n = 0$, we get $RF(<_R) = \ll_R$. Hence $RF(<_R)$ is cycle-free because $<_R$ is a partial order.

Induction steps: Let $n = k$ be an arbitrary number and suppose that $\ll_R \cup r_{1..k}$ is acyclic. When $n = k + 1$, we have $RF(<_R) = \ll_R \cup r_{1..k} \cup \{r_{k+1}\}$. There are two cases for r_{k+1}.

Case 1: r_{k+1} is generated by Φ, which means there exists an element $e \in \ll_R \backslash \ll_C$ such that $r_{k+1} = e.s \mapsto !m$ where $!m \in \rho((\ll_R \cup r_{1..k})^*, e.s, e.d)$. Due to Lemma 1, there will always be an $!m$ such that $!m \not\rightsquigarrow_0 e.s$, so r_{k+1} will not form a cycle.

Case 2: r_{k+1} is generated by Γ, which implies there exists an r_i in $r_{1..k}$ and an input event $?o$ such that $?o \to r_i.d$ and $r_{k+1} = ?o \mapsto !n$ where $!n \in \rho((\ll_R \cup r_{1..k})^*, ?o, r_i.s)$. Similarly, Lemma 1 asserts $!n$ always exists such that $!n \not\rightsquigarrow_0 ?o$. Hence r_{k+1} will not form a cycle.

Since r_{k+1} does not form a cycle in both cases, the graph of $RF(<_R)$ is acyclic. □

The final two propositions justify our intention in this paper. For an inherent refinement ordering $<_R$, we have found a strengthened partial order $<_{RF} = RF(<_R)$ such that both $<_{RF}$ and $IMP(<_{RF})$ are race-free. Note that the application of IMP on $<_{RF}$ requires its variant $\ll_{RF}$, which is actually the relation $\Phi(\ll_R, \ll_R \backslash \ll_C)$ before the transitive closure operator is performed.

Proposition 4. *For an $<_R$, $<_{RF}$ is race-free.*

Proof. We prove by contradiction. Supposing that $<_{RF}$ has race conditions. Since $<_R$ is race-free, for $<_{RF}$ to get a new race condition, the function RF must add an extra edge linking an event $v \in \Sigma$ with an input event $?x$ such that $v \rightarrow ?x$ but $v \not\rightsquigarrow !x$. This contradicts the definition of RF since all the edges added or replaced by RF link an event with an *output* event in the form of $v \rightarrow !x$. □

Proposition 5. *For an $<_R$, $IMP(<_{RF})$ is race-free.*

Proof. We prove by contradiction. Supposing that $IMP(<_{RF})$ is not race-free. The graph of the relation $<_{RF})$ must satisfy either one of the following two cases. We show that both cases contradict the mechanism of RF.

Case 1: $\exists\, r : \ll_{RF} \backslash \ll_C$ such that $v : \Sigma, v \neq r.s \wedge v \rightharpoondown r.d$.
In terms of the graph, this case holds when there are two dashed edges pointing to a single output event. This situation contradicts the definition of Γ no matter v is an input or an output event. If $v \in \Sigma_{out}$, the equation (1) or (5) of Γ applies, both of which eliminate the pair $v \mapsto r.d$ from the relation. If $v \in \Sigma_{in}$, the equation (3) or (4) is applicable, replacing $v \mapsto r.d$ with a different edge.

Case 2: $\exists\, r : \ll_{RF} \backslash \ll_C$ such that $v : \Sigma, v \neq r.s \wedge v \rightharpoonup r.d \wedge v \not\rightsquigarrow r.s$.
This case also contradicts the definition of Γ no matter v is an input or an output event. If $v \in \Sigma_{in}$, either equation (2) or (5) applies, and both equations add another edge so that $v \rightsquigarrow r.s$. If $v \in \Sigma_{out}$, then we have $v \rightsquigarrow r.s$. Otherwise, due to the definition of ρ, the edge r should have linked $r.s$ with v instead of $r.d$ in the first place. □

4.3 Examples

We use a couple of MSC scenarios with a reasonable level of complexity to illustrate our approach. For economy of space, we only show the MSC scenarios of an inherent refinement ordering and its corresponding race-free refinement instead of the partial orders. $MSC4$ in Fig.8 is an MSC with race conditions between three pairs of events, i.e. $(!a, ?c)$, $(!c, ?e)$ and $(?b, ?d)$. The MSC matching its inherent refinement ordering is depicted as $MSC4'$. In $MSC4'$, general ordering symbols are added to delay output events so that the resulting semantics satisfies the criteria of a race-free partial order. Nevertheless, if an acknowledgement message τ is used here to enforce the general ordering $!a \mapsto !c$, we can see that a new race condition arises between $!b$ and $?\tau$. This situation also applies to the other two general orderings $(!c, ?e)$ and $(?b, ?d)$.

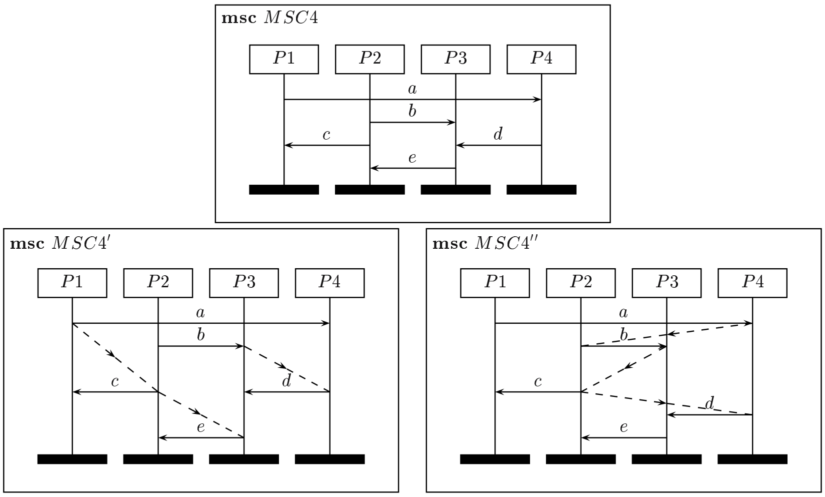


Fig. 8. A simple example

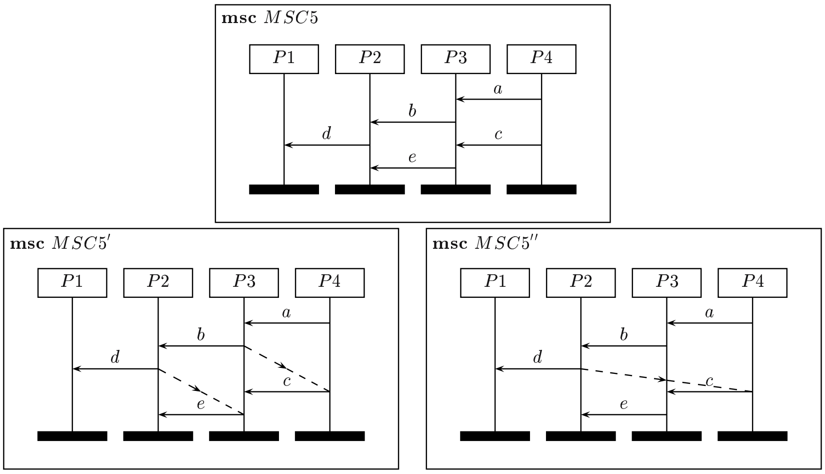


Fig. 9. A simple example

As illustrated beside $MSC4'$, $MSC4''$ in Fig.8 is the matching MSC scenario of the race-free refinement, i.e. $<_{RF}$, that we propose as the solution in this paper. In this case, adding acknowledgement messages to enforce the general orderings does not cause new race conditions.

The other example can be found in Fig.9, where $MSC5$ has race conditions in two places. Two general ordering symbols are added to form its inherent refinement scenario as shown in $MSC5'$. Our solution, however, requires only one general ordering symbol to solve the problem as depicted in $MSC5''$. As to the implementation, $MSC5'$ needs two additional messages, causing new race conditions. Yet $MSC5''$ needs only one, ending up with another race-free scenario.

5 Conclusion

In this paper, we have investigated race conditions in a specification language, namely MSCs, based on partial-order semantics. The existing solution to correcting race conditions in an MSC is a canonical refinement, called inherent refinement ordering, which strengthens the causal ordering of the MSC with aid of general ordering constructs. We claim that new race conditions may occur if specifiers intend to reveal explicitly in MSCs how the general orderings are implemented by adding acknowledgement messages. This observation makes the inherent refinement orderings too weak for solving race conditions in practice.

This paper contributes an approach to finding a minimal strengthening partial order, called race-free refinement, of an inherent refinement ordering. We prove that for an inherent refinement ordering there exists a race-free refinement such that its matching MSC is race-free and remains race-free when acknowledgement messages enforcing general orderings are added. The approach is an algorithm on partial orders. The algorithm is presented in a functional style and can be later implemented for tool-support.

References

1. Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of Message Sequence Charts. *IEEE Transaction on Software Engineering*, 29:623–633, July 2003.
2. Rajeev Alur, Gerard J. Holzmann, and Doron Peled. An analyzer for Message Sequence Charts. *Software Concepts and Tools*, 17(2):70–77, 1996.
3. Hanêne Ben-Abdallah and Stefan Leue. Syntactic detection of process divergence and non-local choice in Message Sequence Charts. In *Proceedings of TACAS'97 (LNCS 1217)*, Netherland, April 1997. Springer-Verlag.
4. Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998. ISBN 0-201-30998-x.
5. Manfred Broy. On the meaning of Message Sequence Charts. In *Proceedings of the 1st Workshop of the SDL Forum Society Workshop on SDL and MSC*, volume I, pages 13–34, 1998.
6. CCITT. *CCITT Recommendation Z.120: Message Sequence Chart (MSC).* Geneva, 1992.
7. C. Chen, S. Kalvala, and J. Sinclair. A process-based semantics for Message Sequence Charts with data. In *Australian Software Engineering Conference 2005 (ASWEC2005)*, Brisbane, March 2005.
8. W. Damm and D. Harel. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
9. Thomas Gehrke, Michaela Huhn, Arend Rensink, and Heike Wehrheim. An algebraic semantics for Message Sequence Charts documents. In *Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV '98)*, Kluwer, 1998.

10. J. Grabowski, P. Graubmann, and E. Rudolph. Towards a Petri net based semantics definition for Message Sequence Charts. In *SDL'93 Using Objects*, Darmstadt, 1993. Proceeding of the 6th SDL Forum.
11. G. J. Holzmann, D. Peled, and M. H. Redberg. Design tools for requirements engineering. *Bell Lab Technical Journal*, 2(1):86–95, 1997.
12. ITU-TS. *Recommendation Z.120: Message Sequence Chart (MSC).* Geneva, 1996.
13. P. B. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.
14. P.B. Ladkin and S. Leue. What do Message Sequence Charts mean? In R.L. Tenney, P.D. Amer, and M.U. Uyar, editors, *Formal Description Techniques VI, IFIP Transactions C*, North-Holland, 1994. Proceeding of the 6th International Conference on Formal Description Techniques.
15. S. Mauw and M.A. Reniers. An algebraic semantics of basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.
16. Bill Mitchell. Inherent causal orderings of partial order scenarios. In *International Colloquim on Theoretical Aspects of Computing (LNCS 3407)*, China, September 2004. Springer-Verlag.
17. Anca Muscholl and Doron Peled. Message sequence graphs and decision problems on mazurkiewicz traces. In *MFCS*, pages 81–91, 1999.
18. A. Olsen, O. Færgemand, B Møller Pedersen, R. Reed, and J.R.W. Smith. *Systems Using SDL-92.* North Holland, 1994.
19. J. Schumann and J. Whittle. Generating statechart designs from scenarios. In *Proceeding of the 22nd International Conference on Software Engineering*, 2000.
20. S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Transaction on Software Engineering*, 29(2), February 2003.

A Next-Generation Platform for Analyzing Executables[⋆]

T. Reps[1,2], G. Balakrishnan[1], J. Lim[1], and T. Teitelbaum[2]

[1] Comp. Sci. Dept., University of Wisconsin,
{reps, bgogul, junghee}@cs.wisc.edu
[2] GrammaTech, Inc.,
{tt}@grammatech.com

Abstract. In recent years, there has been a growing need for tools that an analyst can use to understand the workings of COTS components, plugins, mobile code, and DLLs, as well as memory snapshots of worms and virus-infected code. Static analysis provides techniques that can help with such problems; however, there are several obstacles that must be overcome:

- For many kinds of potentially malicious programs, symbol-table and debugging information is entirely absent. Even if it is present, it cannot be relied upon.
- To understand memory-access operations, it is necessary to determine the set of addresses accessed by each operation. This is difficult because
  - While some memory operations use explicit memory addresses in the instruction (easy), others use indirect addressing via address expressions (difficult).
  - Arithmetic on addresses is pervasive. For instance, even when the value of a local variable is loaded from its slot in an activation record, address arithmetic is performed.
  - There is no notion of type at the hardware level, so address values cannot be distinguished from integer values.
  - Memory accesses do not have to be aligned, so word-sized address values could potentially be cobbled together from misaligned reads and writes.

We have developed static-analysis algorithms to recover information about the contents of memory locations and how they are manipulated by an executable. By combining these analyses with facilities provided by the IDAPro and CodeSurfer toolkits, we have created CodeSurfer/x86, a prototype tool for browsing, inspecting, and analyzing x86 executables. From an x86 executable, CodeSurfer/x86 recovers intermediate representations that are similar to what would be created by a compiler for a program written in a high-level language. CodeSurfer/x86 also supports a scripting language, as well as several kinds of sophisticated pattern-matching capabilities. These facilities provide a platform for the development of additional tools for analyzing the security properties of executables.

1 Introduction

Market forces are increasingly pushing companies to deploy COTS software when possible—for which source code is typically unavailable—and to outsource develop-

[⋆] Portions of this paper have appeared in [3,4].

ment when custom software is required. Moreover, a great deal of legacy code—for which design documents are usually out-of-date, and for which source code is sometimes unavailable and sometimes non-existent—will continue to be left deployed. An important challenge during the coming decade will be how to identify bugs and security vulnerabilities in such systems. Methods are needed to determine whether third-party and legacy application programs can perform malicious operations (or can be induced to perform malicious operations), and to be able to make such judgments in the absence of source code.

Recent research in programming languages, software engineering, and computer security has led to new kinds of tools for analyzing code for bugs and security vulnerabilities [25,40,20,14,8,5,10,27,17,9]. In these tools, static analysis is used to determine a conservative answer to the question "Can the program reach a bad state?"[1] In principle, such tools would be of great help to an analyst trying to detect malicious code hidden in software, except for one important detail: the aforementioned tools all focus on analyzing *source code* written in a high-level language. Even if source code were available, there are a number of reasons why analyses that start from source code do not provide the right level of detail for checking certain kinds of properties, which can cause bugs, security vulnerabilities, and malicious behavior to be invisible to such tools. (See §2.)

In contrast, our work addresses the problem of finding bugs and security vulnerabilities in programs when source code is unavailable. Our goal is to create a platform that carries out static analysis on executables and provides information that an analyst can use to understand the workings of potentially malicious code, such as COTS components, plugins, mobile code, and DLLs, as well as memory snapshots of worms and virus-infected code. A second goal is to use this platform to create tools that an analyst can employ to determine such information as

– whether a program contains inadvertent security vulnerabilities
– whether a program contains deliberate security vulnerabilities, such as back doors, time bombs, or logic bombs. If so, the goal is to provide information about activation mechanisms, payloads, and latencies.

We have developed a tool, called CodeSurfer/x86, that serves as a prototype for a next-generation platform for analyzing executables. CodeSurfer/x86 provides a security analyst with a powerful and flexible platform for investigating the properties and possible behaviors of an x86 executable. It uses static analysis to recover intermediate representations (IRs) that are similar to those that a compiler creates for a program written in a high-level language. An analyst is able to use (i) CodeSurfer/x86's GUI, which provides mechanisms to understand a program's chains of data and control dependences, (ii) CodeSurfer/x86's scripting language, which provides access to all of the intermediate representations that CodeSurfer/x86 builds, and (iii) GrammaTech's Path Inspector, which is a model-checking tool that uses a sophisticated pattern-matching engine to answer questions about the flow of execution in a program.

[1] Static analysis provides a way to obtain information about the possible states that a program reaches during execution, but without actually running the program on specific inputs. Static-analysis techniques explore the program's behavior for *all* possible inputs and *all* possible states that the program can reach. To make this feasible, the program is "run in the aggregate"—i.e., on descriptors that represent *collections* of memory configurations [15].

Because CodeSurfer/x86 was designed to provide a platform that an analyst can use to understand the workings of potentially malicious code, a major challenge is that the tool must assume that the x86 executable is untrustworthy, and hence symbol-table and debugging information cannot be relied upon (even if it is present). The algorithms used in CodeSurfer/x86 provide ways to meet this challenge.

Although the present version of CodeSurfer/x86 is targeted to x86 executables, the techniques used [3,34,37,32] are language-independent and could be applied to other types of executables. In addition, it would be possible to extend CodeSurfer/x86 to use symbol-table and debugging information in situations where such information is available and trusted—for instance, if you have the source code for the program, you invoke the compiler yourself, and you trust the compiler to supply correct symbol-table and debugging information. Moreover, the techniques extend naturally if source code is available: one can treat the executable code as just another IR in the collection of IRs obtainable from source code. The mapping of information back to the source code would be similar to what C source-code tools already have to perform because of the use of the C preprocessor (although the kind of issues that arise when debugging optimized code [26,43,16] complicate matters).

The remainder of paper is organized as follows: §2 illustrates some of the advantages of analyzing executables. §3 describes CodeSurfer/x86. §4 gives an overview of the model-checking facilities that have been coupled to CodeSurfer/x86. §5 discusses related work.

2 Advantages of Analyzing Executables

This section discusses why an analysis that works on executables can provide more accurate information than an analysis that works on source code.[2] An analysis that works on source code can fail to detect certain bugs and vulnerabilities due to the WYSINWYX phenomenon: “What You See Is Not What You eXecute” [4], which can cause there to be a mismatch between what a programmer intends and what is actually executed by the processor. The following source-code fragment, taken from a login program, illustrates the issue [29]:

```
    memset(password, '\0', len);
    free(password);
```

The login program temporarily stores the user’s password—in clear text—in a dynamically allocated buffer pointed to by the pointer variable `password`. To minimize the lifetime of the password, which is sensitive information, the code fragment shown above zeroes-out the buffer pointed to by `password` before returning it to the heap. Unfortunately, a compiler that performs useless-code elimination may reason that the program never uses the values written by the call on `memset`, and therefore the call on memset can be removed—thereby leaving sensitive information exposed in the heap. This is not just hypothetical; a similar vulnerability was discovered during the Windows security push in 2002 [29]. This vulnerability is invisible in the source code; it can only be detected by examining the low-level code emitted by the optimizing compiler.

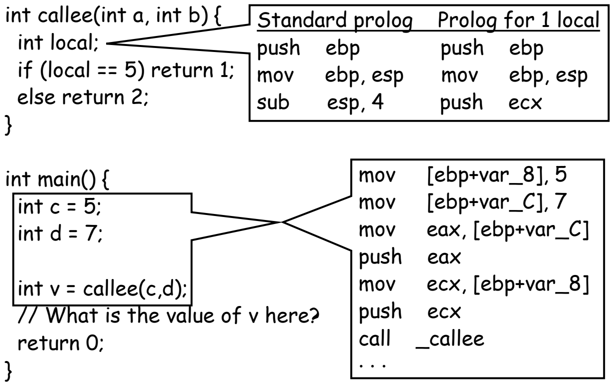
[2] Terms like “an analysis that works on source code” and “source-level analyses” are used as a shorthand for “analyses that work on IRs built from the source code.”

A second example where analysis of an executable does better than typical source-level analyses involves pointer arithmetic and an indirect call:

```
int (*f)(void);
int diff=(char*)&f2-(char*)&f1;//Theoffsetbetweenf1andf2
f = &f1;
f = (int (*)())((char*)f + diff); // f now points to f2
(*f)(); // indirect call;
```

Existing source-level analyses (that we know of) are ill-prepared to handle the above code. The conventional assumption is that arithmetic on function pointers leads to undefined behavior, so source-level analyses either (a) assume that the indirect function call might call any function, or (b) ignore the arithmetic operations and assume that the indirect function call calls `f1` (on the assumption that the code is ANSI-C compliant). In contrast, the analysis described by Balakrishnan and Reps [3] correctly identifies `f2` as the invoked function. Furthermore, the analysis can detect when arithmetic on addresses creates an address that does not point to the beginning of a function; the use of such an address to perform a function "call" is likely to be a bug (or else a very subtle, deliberately introduced security vulnerability).

A third example involves a function call that passes fewer arguments than the procedure expects as parameters. (Many compilers accept such (unsafe) code as an easy way of implementing functions that take a variable number of parameters.) With most compilers, this effectively means that the call-site passes some parts of one or more local variables of the calling procedure as the remaining parameters (and, in effect, these are passed by reference—an assignment to such a parameter in the callee will overwrite the value of the corresponding local in the caller.) An analysis that works on executables can be created that is capable of determining what the extra parameters are [3], whereas a source-level analysis must either make a cruder over-approximation or an unsound under-approximation.



```
int callee(int a, int b) {
  int local;
  if (local == 5) return 1;
  else return 2;
}
```

Standard prolog		Prolog for 1 local	
push	ebp	push	ebp
mov	ebp, esp	mov	ebp, esp
sub	esp, 4	push	ecx

```
int main() {
  int c = 5;
  int d = 7;

  int v = callee(c,d);
  // What is the value of v here?
  return 0;
}
```

```
mov    [ebp+var_8], 5
mov    [ebp+var_C], 7
mov    eax, [ebp+var_C]
push   eax
mov    ecx, [ebp+var_8]
push   ecx
call   _callee
. . .
```

Fig. 1. Example of unexpected behavior due to compiler optimization. The box at the top right shows two variants of code generated by an optimizing compiler for the prolog of `callee`. Analysis of the second of these reveals that the variable `local` necessarily contains the value 5.

A final example is shown in Fig. 1. The C code on the left uses an uninitialized variable (which triggers a compiler warning, but compiles successfully). A source-code analyzer must assume that `local` can have any value, and therefore the value of `v` in `main` is either 1 or 2. The assembly listings on the right show how the C code could be compiled, including two variants for the prolog of function `callee`. The Microsoft compiler (cl) uses the second variant, which includes the following strength reduction:

> *The instruction* `sub esp,4` *that allocates space for* `local` *is replaced by a* `push` *instruction of an arbitrary register (in this case,* `ecx`*).*

An analysis of the executable can determine that this optimization results in `local` being initialized to 5, and therefore `v` in `main` can only have the value 1.

To summarize, the advantage of an analysis that works on executables is that an executable contains the actual instructions that will be executed, and hence provides information that reveals the actual behavior that arises during program execution. This information includes

- memory-layout details, such as (i) the positions (i.e., offsets) of variables in the runtime stack's activation records, and (ii) padding between structure fields.
- register usage
- execution order (e.g., of actual parameters)
- optimizations performed
- artifacts of compiler bugs

Access to such information can be crucial; for instance, many security exploits depend on platform-specific features, such as the structure of activation records. Vulnerabilities can escape notice when a tool does not have information about adjacency relationships among variables.

In contrast, there are a number of reasons why analyses based on source code do not provide the right level of detail for checking certain kinds of properties:

- Source-level tools are only applicable when source is available, which limits their usefulness in security applications (e.g., to analyzing code from open-source projects).
- Analyses based on source code typically make (unchecked) assumptions, e.g., that the program is ANSI-C compliant. This often means that an analysis does not account for behaviors that are allowed by the compiler (e.g., arithmetic is performed on pointers that are subsequently used for indirect function calls; pointers move off the ends of arrays and are subsequently dereferenced; etc.)
- Programs typically make extensive use of libraries, including dynamically linked libraries (DLLs), which may not be available in source-code form. Typically, source-level analyses are performed using code stubs that model the effects of library calls. Because these are created by hand they are likely to contain errors, which may cause an analysis to return incorrect results.
- Programs are sometimes modified subsequent to compilation, e.g., to perform optimizations or insert instrumentation code [41]. (They may also be modified to insert malicious code.) Such modifications are not visible to tools that analyze source.
- The source code may have been written in more than one language. This complicates the life of designers of tools that analyze source code because multiple languages must be supported, each with its own quirks.

– Even if the source code is primarily written in one high-level language, it may contain inlined assembly code in selected places. Source-level analysis tools typically either skip over inlined assembly code [13] or do not push the analysis beyond sites of inlined assembly code [1].

Thus, even if source code is available, a substantial amount of information is hidden from analyses that start from source code, which can cause bugs, security vulnerabilities, and malicious behavior to be invisible to such tools. Moreover, a source-level analysis tool that strives to have greater fidelity to the program that is actually executed would have to duplicate all of the choices made by the compiler and optimizer; such an approach is doomed to failure.

3 Analyzing Executables in the Absence of Source Code

To be able to apply techniques like the ones used in [25,40,20,14,8,5,10,27,17,9], one already encounters a challenging program-analysis problem. From the perspective of the compiler community, one would consider the problem to be "IR recovery": one needs to recover *intermediate representations* from the executable that are similar to those that would be available had one started from source code. From the perspective of the model-checking community, one would consider the problem to be that of "model extraction": one needs to extract a suitable *model* from the executable. To solve the IR-recovery problem, several obstacles must be overcome:

– For many kinds of potentially malicious programs, symbol-table and debugging information is entirely absent. Even if it is present, it cannot be relied upon.
– To understand memory-access operations, it is necessary to determine the set of addresses accessed by each operation. This is difficult because
  • While some memory operations use explicit memory addresses in the instruction (easy), others use indirect addressing via address expressions (difficult).
  • Arithmetic on addresses is pervasive. For instance, even when the value of a local variable is loaded from its slot in an activation record, address arithmetic is performed.
  • There is no notion of type at the hardware level, so address values cannot be distinguished from integer values.
  • Memory accesses do not have to be aligned, so word-sized address values could potentially be cobbled together from misaligned reads and writes.

For the past few years, we have been working to create a prototype next-generation platform for analyzing executables. The tool set that we have developed extends static vulnerability-analysis techniques to work directly on executables, even in the absence of source code. The tool set builds on (i) recent advances in static analysis of program executables [3], and (ii) new techniques for software model checking and dataflow analysis [7,36,37,32]. The main components of the tool set are *CodeSurfer/x86*, *WPDS++*, and the *Path Inspector*:

– CodeSurfer/x86 recovers IRs from an executable that are similar to the IRs that source-code-analysis tools create—but, in many respects, the IRs that

CodeSurfer/x86 builds are more precise. CodeSurfer/x86 also provides an API to these IRs.
- WPDS++ [31] is a library for answering generalized reachability queries on *weighted pushdown systems* (WPDSs) [7,36,37,32]. This library provide a mechanism for defining and solving model-checking and dataflow-analysis problems. To extend CodeSurfer/x86's analysis capabilities, the CodeSurfer/x86 API can be used to extract a WPDS model from an executable and to run WPDS++ on the model.
- The Path Inspector is a software model checker built on top of CodeSurfer and WPDS++. It supports safety queries about a program's possible control configurations.

In addition, by writing scripts that traverse the IRs that CodeSurfer/x86 recovers, the tool set can be extended with further capabilities (e.g., decompilation, code rewriting, etc.).

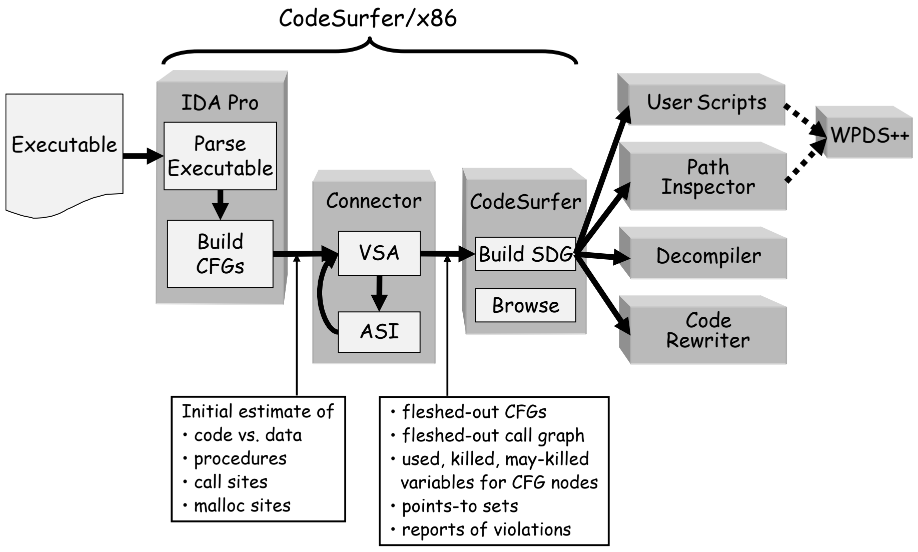


Fig. 2. Organization of CodeSurfer/x86 and companion tools

Fig. 2 shows how these components fit together. CodeSurfer/x86 makes use of both IDAPro [30], a disassembly toolkit, and GrammaTech's CodeSurfer system [13], a toolkit originally developed for building program-analysis and inspection tools that analyze source code. These components are glued together by a piece called the Connector, which uses two static analyses—value-set analysis (VSA) [3] and aggregate-structure identification (ASI) [34] to recover information about the contents of memory locations and how they are manipulated by an executable.[3]

[3] VSA also makes use of the results of an additional static-analysis phase, called affine-relation analysis (ARA), which, for each program point, identifies affine relationships [33] that hold among the values of registers; see [3,32].

An x86 executable is first disassembled using IDAPro. In addition to the disassembly listing, IDAPro also provides access to the following information:

Statically known memory addresses and offsets: IDAPro identifies the statically known memory addresses and stack offsets in the program, and renames all occurrences of these quantities with a consistent name. This database is used to define the set of data objects in terms of which (the initial run of) VSA is carried out; these objects are called *a-locs*, for "abstract locations". VSA is an analysis that, for each instruction, determines an over-approximation of the set of values that each a-loc could hold.

Information about procedure boundaries: X86 executables do not have information about procedure boundaries. IDAPro identifies the boundaries of most of the procedures in an executable.[4]

Calls to library functions: IDAPro discovers calls to library functions using an algorithm called Fast Library Identification and Recognition Technology (FLIRT) [23].

IDAPro provides access to its internal resources via an API that allows users to create plug-ins to be executed by IDAPro. CodeSurfer/x86 uses a plug-in to IDAPro, called the Connector, that creates data structures to represent the information that it obtains from IDAPro (see Fig. 2); VSA and ASI are implemented using the data structures created by the Connector. The IDAPro/Connector combination is also able to create the same data structures for DLLs, and to link them into the data structures that represent the program itself. This infrastructure permits whole-program analysis to be carried out—including analysis of the code for all library functions that are called.

CodeSurfer/x86 makes no use of symbol-table or debugging information. Instead, the results of VSA and ASI provide a substitute for absent or untrusted symbol-table and debugging information. Initially, the set of a-locs is determined based on the static memory addresses and stack offsets that are used in instructions in the executable. Each run of ASI refines the set of a-locs used for the next run of VSA.

Because the IRs that CodeSurfer/x86 recovers are extracted directly from the executable code that is run on the machine, and because the entire program is analyzed—including any libraries that are linked to the program—this approach provides a "higher fidelity" platform for software model checking than the IRs derived from source code that other software model checkers use [25,40,20,14,8,5,10,27,17,9].

CodeSurfer/x86 supports a scripting language that provides access to all of the IRs that CodeSurfer/x86 builds for the executable. This provides a way to connect CodeSurfer/x86 to other analysis tools, such as model checkers (see §4), as well as to implement other tools on top of CodeSurfer/x86, such as decompilers, code rewriters, etc. It also provides an analyst with a mechanism to develop any additional "one-off" analyses he needs to create.

[4] IDAPro does not identify the targets of all indirect jumps and indirect calls, and therefore the call graph and control-flow graphs that it constructs are not complete. However, the information computed during VSA is used to augment the call graph and control-flow graphs on-the-fly to account for indirect jumps and indirect calls.

3.1 Memory-Access Analysis in the Connector

The analyses in CodeSurfer/x86 are a great deal more ambitious than even relatively sophisticated disassemblers, such as IDAPro. At the technical level, CodeSurfer/x86 addresses the following problem:

Given a stripped executable E, identify the
- procedures, data objects, types, and libraries that it uses

and
- for each instruction I in E and its libraries
- for each interprocedural calling context of I
- for each machine register and a-loc A

statically compute an accurate over-approximation to
- the set of values that A may contain when I executes
- the instructions that may have defined the values used by I
- the instructions that may use the values defined by execution of I

and provide effective means to access that information both interactively and under program control.

Value-Set Analysis. VSA [3] is a combined numeric and pointer-analysis algorithm that determines an over-approximation of the set of numeric values and addresses (or *value set*) that each a-loc holds at each program point. The information computed during VSA is used to augment the call graph and control-flow graphs on-the-fly to account for indirect jumps and indirect function calls.

VSA is related to pointer-analysis algorithms that have been developed for programs written in high-level languages, which determine an over-approximation of the set of variables whose addresses each pointer variable can hold:

> *VSA determines an over-approximation of the set of addresses that each data object can hold at each program point.*

At the same time, VSA is similar to range analysis and other numeric static-analysis algorithms that over-approximate the integer values that each variable can hold:

> *VSA determines an over-approximation of the set of integer values that each data object can hold at each program point.*

The following insights shaped the design of VSA:

- A *non-aligned access* to memory—e.g., an access via an address that is not aligned on a 4-byte word boundary—spans parts of two words, and provides a way to forge a new address from *parts* of old addresses. It is important for VSA to discover information about the alignments and strides of memory accesses, or else most indirect-addressing operations appear to be possibly non-aligned accesses.
- To prevent most loops that traverse arrays from appearing to be possible stack-smashing attacks, the analysis needs to use relational information so that the values of a-locs assigned to within a loop can be related to the values of the a-locs used in the loop's branch condition (see [3,33,32]).

– It is desirable for VSA to track integer-valued and address-valued quantities *simultaneously*. This is crucial for analyzing executables because
  - integers and addresses are indistinguishable at execution time, and
  - compilers use address arithmetic and indirect addressing to implement such features as pointer arithmetic, pointer dereferencing, array indexing, and accessing structure fields.

  Moreover, information about integer values can lead to improved tracking of address-valued quantities, and information about address values can lead to improved tracking of integer-valued quantities.

VSA produces information that is more precise than that obtained via several more conventional numeric analyses used in compilers, including constant propagation, range analysis, and integer-congruence analysis. At the same time, VSA provides an analog of pointer analysis that is suitable for use on executables.

Aggregate-Structure Identification. One of the major stumbling blocks in analysis of executables is the difficulty of recovering information about variables and types, especially for aggregates (i.e., structures and arrays). CodeSurfer/x86 uses an iterative strategy for recovering such information; with each round, it refines its notion of the program's variables and types.

Initially, VSA uses a set of variables ("a-locs") that are obtained from IDAPro. Because IDAPro has relatively limited information available at the time that it applies its variable-discovery heuristics (i.e., it only knows about statically known memory addresses and stack offsets), what it can do is rather limited, and generally leads to a very coarse-grained approximation of the program's variables.

Once a given run of VSA completes, the value-sets for the a-locs at each instruction provide a way to identify an over-approximation of the memory accesses performed at that instruction. This information is used to refine the current set of a-locs by running a variant of the ASI algorithm [34], which identifies commonalities among accesses to different parts of an aggregate data value. ASI was originally developed for analysis of Cobol programs: in that context, ASI ignores all of the type declarations in the program, and considers an aggregate to be merely a sequence of bytes of a given length; an aggregate is then broken up into smaller parts depending upon how the aggregate is accessed by the program. In the context in which we use ASI—namely, analysis of x86 executables—ASI cannot be applied until the results of VSA are already in hand: ASI requires points-to, range, and stride information to be available; however, for an x86 executable this information is not available until after VSA has been run.

ASI exploits the information made available by VSA (such as the values that a-locs can hold, sizes of arrays, and iteration counts for loops), which generally leads to a much more accurate set of a-locs than the initial set of a-locs discovered by IDAPro. For instance, consider a simple loop, implemented in source code as

```
int a[10], i;
for (i = 0; i < 10; i++)
   a[i] = i;
```

From the executable, IDAPro will determine that there are two variables, one of size 4 bytes and one of size 40 bytes, but will provide no information about the substructure of the 40-byte variable. In contrast, in addition to the 4-byte variable, ASI will correctly identify that the 40 bytes are an array of ten 4-byte quantities.

The Connector uses a refinement loop that performs repeated phases of VSA and ASI (see Fig. 2). The ASI results are used to refine the previous set of a-locs, and the refined set of a-locs is then used to analyze the program during the next round of VSA. The number of iterations is controlled by a command-line parameter.

ASI also provides information that greatly increases the precision with which VSA can analyze the contents of dynamically allocated objects (i.e., memory locations allocated using malloc or new). To see why, recall how the initial set of a-locs is identified by IDAPro. The a-loc abstraction exploits the fact that accesses to program variables in a high-level language are either complied into static addresses (for globals, and fields of struct-valued globals) or static stack-frame offsets (for locals and fields of struct-valued locals). However, fields of dynamically allocated objects are accessed in terms of offsets relative to the base address of the object itself, which is something that IDAPro knows nothing about. In contrast, VSA considers each malloc site m to be a "memory region" (consisting of the objects allocated at m), and the memory region for m serves as a representative for the base addresses of those objects. This lets ASI handle the use of an offset from an object's base address similar to the way that it handles a stack-frame offset—with the net result that ASI is able to capture information about the fine-grained structure of dynamically allocated objects. The object fields discovered in this way become a-locs for the next round of VSA, which will then discover an over-approximation of their contents.

ASI is complementary to VSA: ASI addresses only the issue of identifying the structure of aggregates, whereas VSA addresses the issue of (over-approximating) the contents of memory locations. ASI provides an improved method for the "variable-identification" facility of IDAPro, which uses only much cruder techniques (and only takes into account statically known memory addresses and stack offsets). Moreover, ASI requires more information to be on hand than is available in IDAPro (such as the sizes of arrays and iteration counts for loops). Fortunately, this is exactly the information that is available after VSA has been carried out, which means that ASI can be used in conjunction with VSA to obtain improved results: after a first round of VSA, the results of ASI are used to refine the a-loc abstraction, after which VSA is run again—generally producing more precise results.

3.2 CodeSurfer/x86

The value-sets for the a-locs at each program point are used to determine each point's sets of used, killed, and possibly-killed a-locs; these are emitted in a format that is suitable for input to CodeSurfer.

CodeSurfer is a tool for code understanding and code inspection that supports both a graphical user interface (GUI) and an API (as well as a scripting language) to provide access to a program's system dependence graph (SDG) [28], as well as other information stored in CodeSurfer's IRs.[5] An SDG consists of a set of program dependence graphs (PDGs), one for each procedure in the program. A vertex in a PDG corresponds to a construct in the program, such as an instruction, a call to a procedure, an actual parameter of a call, or a formal parameter of a procedure. The edges correspond to data and control dependences between the vertices [21]. The PDGs are connected together with interprocedural edges that represent control dependences between procedure calls and entries, data dependences between actual parameters and formal parameters, and data dependences between return values and receivers of return values.

Dependence graphs are invaluable for many applications, because they highlight chains of dependent instructions that may be widely scattered through the program. For example, given an instruction, it is often useful to know its *data-dependence predecessors* (instructions that write to locations read by that instruction) and its *control-dependence predecessors* (control points that may affect whether a given instruction gets executed). Similarly, it may be useful to know for a given instruction its *data-dependence successors* (instructions that read locations written by that instruction) and *control-dependence* successors (instructions whose execution depends on the decision made at a given control point).

CodeSurfer's GUI supports browsing ("surfing") of an SDG, along with a variety of operations for making queries about the SDG—such as slicing [28] and chopping [35].[6]

The GUI allows a user to navigate through a program's source code using these dependences in a manner analogous to navigating the World Wide Web.

CodeSurfer's API provides a programmatic interface to these operations, as well as to lower-level information, such as the individual nodes and edges of the program's SDG, call graph, and control-flow graph, and a node's sets of used, killed, and possibly-killed a-locs. By writing programs that traverse CodeSurfer's IRs to implement additional program analyses, the API can be used to extend CodeSurfer's capabilities.

CodeSurfer/x86 provides some unique capabilities for answering an analyst's questions. For instance, given a worm, CodeSurfer/x86's analysis results have been used to obtain information about the worm's target-discovery, propagation, and activation mechanisms by

- locating sites of system calls,
- finding the instructions by which arguments are passed, and
- following dependences backwards from those instructions to identify where the values come from.

[5] In addition to the SDG, CodeSurfer's IRs include abstract-syntax trees, control-flow graphs (CFGs), a call graph, VSA results, the sets of used, killed, and possibly killed a-locs at each instruction, and information about the structure and layout of global memory, activation records, and dynamically allocated storage.

[6] A backward slice of a program with respect to a set of program points S is the set of all program points that might affect the computations performed at S; a forward slice with respect to S is the set of all program points that might be affected by the computations performed at members of S [28]. A program chop between a set of source program points S and a set of target program points T shows how S can affect the points in T [35]. Chopping is a key operation in information-flow analysis.

Because the techniques described in §3.1 are able to recover quite rich information about memory-access operations, the answers that CodeSurfer/x86 furnishes to such questions account for the movement of data through memory—not just the movement of data through registers, as in some prior work (e.g., [18,11]).

3.3 Goals, Capabilities, and Assumptions

A few words are in order about the goals, capabilities, and assumptions underlying CodeSurfer/x86.

The constraint that symbol-table and debugging information are off-limits complicated the task of creating CodeSurfer/x86; however, the results of VSA and ASI provide a substitute for such information. This allowed us to create a tool that can be used when symbol-table and debugging information is absent or untrusted.

Given an executable as input, the goal is to check whether the executable conforms to a "standard" compilation model—i.e., a runtime stack is maintained; activation records (ARs) are pushed onto the stack on procedure entry and popped from the stack on procedure exit; each global variable resides at a fixed offset in memory; each local variable of a procedure f resides at a fixed offset in the ARs for f; actual parameters of f are pushed onto the stack by the caller so that the corresponding formal parameters reside at fixed offsets in the ARs for f; the program's instructions occupy a fixed area of memory, are not self-modifying, and are separate from the program's data. If the executable conforms to this model, CodeSurfer/x86 creates an IR for it. If it does not conform to the model, then one or more violations will be discovered, and corresponding error reports are issued.

The goal for CodeSurfer/x86 is to provide (i) a tool for security analysis, and (ii) a general infrastructure for additional analysis of executables. Thus, as a practical measure, when the system produces an error report, a choice is made about how to accommodate the error so that analysis can continue (i.e., the error is optimistically treated as a false positive), and an IR is produced; if the analyst can determine that the error report is indeed a false positive, then the IR is valid.

The analyzer does not care whether the program was compiled from a high-level language, or hand-written in assembly code. In fact, some pieces of the program may be the output from a compiler (or from multiple compilers, for different high-level languages), and others hand-written assembly code. Still, it is easiest to talk about the information that VSA and ASI are capable of recovering in terms of the features that a high-level programming language allows: VSA and ASI are capable of recovering information from programs that use global variables, local variables, pointers, structures, arrays, heap-allocated storage, pointer arithmetic, indirect jumps, recursive procedures, indirect calls through function pointers, virtual-function calls, and DLLs (but, at present, not run-time code generation or self-modifying code).

Compiler optimizations often make VSA and ASI *less* difficult, because more of the computation's critical data resides in registers, rather than in memory; register operations are more easily deciphered than memory operations.

The major assumption that we make about IDAPro is that it is able to disassemble a program and build an adequate collection of *preliminary* IRs for it. Even though (i) the CFG created by IDAPro may be incomplete due to indirect jumps, and (ii) the call-graph created by IDAPro may be incomplete due to indirect calls, incomplete IRs do *not* trigger error reports. Both the CFG and the call-graph are fleshed out according to information recovered during the course of VSA/ASI iteration. In fact, the relationship between VSA/ASI iteration and the preliminary IRs created by IDAPro is similar to the relationship between a points-to-analysis algorithm in a C compiler and the preliminary IRs created by the C compiler's front end. In both cases, the preliminary IRs are fleshed out during the course of analysis.

4 Model-Checking Facilities

Model checking [12] involves the use of sophisticated pattern-matching techniques to answer questions about the flow of execution in a program: a model of the program's possible behavior is created and checked for conformance with a model of expected behavior (as specified by a user query). In essence, model-checking algorithms explore the program's state-space and answer questions about whether a bad state can be reached during an execution of the program.

For model checking, the CodeSurfer/x86 IRs are used to build a *weighted pushdown system* (WPDS) [7,36,37,32] that models possible program behaviors. WPDSs generalize a model-checking technology known as *pushdown systems* (PDSs) [6,22], which have been used for software model checking in the Moped [39,38] and MOPS [10] systems. Compared to ordinary (unweighted) PDSs, WPDSs are capable of representing more powerful kinds of abstractions of runtime states [37,32], and hence go beyond the capabilities of PDSs. For instance, the use of WPDSs provides a way to address certain kinds of security-related queries that cannot be answered by MOPS.

WPDS++ [31] is a library that implements the symbolic algorithms from [37,32] for solving WPDS reachability problems. We follow the standard approach of using a PDS to model the interprocedural CFG (one of CodeSurfer/x86's IRs). The stack symbols correspond to program locations; there is only a single PDS state; and PDS rules encode control flow as follows:

Rule	Control flow modeled
$q\langle u\rangle \hookrightarrow q\langle v\rangle$	Intraprocedural CFG edge $u \rightarrow v$
$q\langle c\rangle \hookrightarrow q\langle entry_P\ r\rangle$	Call to P from c that returns to r
$q\langle x\rangle \hookrightarrow q\langle\rangle$	Return from a procedure at exit node x

In a configuration of the PDS, the symbol at the top of the stack corresponds to the current program location, and the rest of the stack holds return-site locations—this allows the PDS to model the behavior of the program's runtime execution stack.

An encoding of the interprocedural CFG as a PDS is sufficient for answering queries about reachable control states (as the Path Inspector does; see below): the reachability algorithms of WPDS++ can determine if an undesirable PDS configuration is reachable. However, WPDS++ also supports *weighted* PDSs, which are PDSs in which each rule is weighted with an element of a (user-defined) semiring. The use of weights allows WPDS++ to perform interprocedural dataflow analysis by using the semiring's *extend* operator to compute weights for sequences of rule firings and using the semiring's *combine* operator to take the meet of weights generated by different paths [37,32]. (When the weights on rules are conservative abstract data transformers, an over-approximation to the set of reachable concrete configurations is obtained, which means that counterexamples reported by WPDS++ may actually be infeasible.)

The advantage of answering reachability queries on WPDSs over conventional dataflow-analysis methods is that the latter merge together the values for all states associated with the same program point, regardless of the states' calling context. With WPDSs, queries can be posed with respect to a regular language of stack configurations [7,36,37,32]. (Conventional merged dataflow information can also be obtained [37].)

CodeSurfer/x86 can also be used in conjunction with GrammaTech's Path Inspector tool. The Path Inspector provides a user interface for automating safety queries that are only concerned with the possible control configurations that an executable can reach. The Path Inspector checks *sequencing properties* of events in a program, which can be used to answer such questions as "Is it possible for the program to bypass the authentication routine?" (which indicates that the program may contain a trapdoor), or "Can this *login* program bypass the code that writes to the log file?" (which indicates that the program may be a Trojan *login* program).

With the Path Inspector, such questions are posed as questions about the existence of problematic event sequences; after checking the query, if a problematic path exists, it is displayed in the Path Inspector tool. This lists all of the program points that may occur along the problematic path. These items are linked to the source code; the analyst can navigate from a point in the path to the corresponding source-code element. In addition, the Path Inspector allows the analyst to step forward and backward through the path, while simultaneously stepping through the source code. (The code-stepping operations are similar to the single-stepping operations in a traditional debugger.)

The Path Inspector uses an automaton-based approach to model checking: the query is specified as a finite automaton that captures forbidden sequences of program locations. This "query automaton" is combined with the program model (a WPDS) using a cross-product construction, and the reachability algorithms of WPDS++ are used to determine if an error configuration is reachable. If an error configuration is reachable, then *witnesses* (see [37]) can be used to produce a program path that drives the query automaton to an error state.

The Path Inspector includes a GUI for instantiating many common reachability queries [19], and for displaying counterexample paths in the disassembly listing. In the current implementation, transitions in the query automaton are triggered by program points that the user specifies either manually, or using result sets from CodeSurfer queries. Future versions of the Path Inspector will support more sophisticated queries in which transitions are triggered by matching an AST pattern against a program location, and query states can be instantiated based on pattern bindings.

5 Related Work

Previous work on analyzing memory accesses in executables has dealt with memory accesses very conservatively: generally, if a register is assigned a value from memory, it is assumed to take on any value. VSA does a much better job than previous work because it tracks the integer-valued and address-valued quantities that the program's data objects can hold; in particular, VSA tracks the values of data objects other than just the hardware registers, and thus is not forced to give up all precision when a load from memory is encountered.

The basic goal of the algorithm proposed by Debray et al. [18] is similar to that of VSA: for them, it is to find an over-approximation of the set of values that each *register* can hold at each program point; for us, it is to find an over-approximation of the set of values that each (abstract) data object can hold at each program point, where data objects include *memory locations* in addition to registers. In their analysis, a set of addresses is approximated by a set of congruence values: they keep track of only the low-order bits of addresses. However, unlike VSA, their algorithm does not make any effort to track values that are not in registers. Consequently, they lose a great deal of precision whenever there is a load from memory.

Cifuentes and Fraboulet [11] give an algorithm to identify an intraprocedural slice of an executable by following the program's use-def chains. However, their algorithm also makes no attempt to track values that are not in registers, and hence cuts short the slice when a load from memory is encountered.

The two pieces of work that are most closely related to VSA are the algorithm for data-dependence analysis of assembly code of Amme et al. [2] and the algorithm for pointer analysis on a low-level intermediate representation of Guo et al. [24]. The algorithm of Amme et al. performs only an *intra*procedural analysis, and it is not clear whether the algorithm fully accounts for dependences between memory locations. The algorithm of Guo et al. [24] is only partially flow-sensitive: it tracks registers in a flow-sensitive manner, but treats memory locations in a flow-insensitive manner. The algorithm uses partial transfer functions [42] to achieve context-sensitivity. The transfer functions are parameterized by "unknown initial values" (UIVs); however, it is not clear whether the the algorithm accounts for the possibility of called procedures corrupting the memory locations that the UIVs represent.

References

1. PREfast with driver-specific rules, October 2004. WHDC, Microsoft Corp., http://www.microsoft.com/whdc/devtools/tools/PREfast-drv.mspx.
2. W. Amme, P. Braun, E. Zehendner, and F. Thomasset. Data dependence analysis of assembly code. *Int. J. Parallel Proc.*, 2000.
3. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Comp. Construct.*, pages 5–23, 2004.
4. G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *IFIP Working Conf. on Verified Software: Theories, Tools, Experiments*, 2005.
5. T. Ball and S.K. Rajamani. The SLAM toolkit. In *Computer Aided Verif.*, volume 2102 of *Lec. Notes in Comp. Sci.*, pages 260–264, 2001.
6. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proc. CONCUR*, volume 1243 of *Lec. Notes in Comp. Sci.*, pages 135–150. Springer-Verlag, 1997.
7. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Princ. of Prog. Lang.*, pages 62–73, 2003.

8. W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software–Practice&Experience*, 30:775–802, 2000.
9. H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Network and Dist. Syst. Security*, 2004.
10. H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Conf. on Comp. and Commun. Sec.*, pages 235–244, November 2002.
11. C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *Int. Conf. on Softw. Maint.*, pages 188–195, 1997.
12. E.M. Clarke, Jr., O. Grumberg, and D.A. Peled. *Model Checking*. The M.I.T. Press, 1999.
13. CodeSurfer, GrammaTech, Inc., http://www.grammatech.com/products/codesurfer/.
14. J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Int. Conf. on Softw. Eng.*, pages 439–448, 2000.
15. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Princ. of Prog. Lang.*, pages 238–252, 1977.
16. D.S. Coutant, S. Meloy, and M. Ruscetta. DOC: A practical approach to source-level debugging of globally optimized code. In *Prog. Lang. Design and Impl.*, 1988.
17. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Prog. Lang. Design and Impl.*, pages 57–68, New York, NY, 2002. ACM Press.
18. S.K. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Princ. of Prog. Lang.*, pages 12–24, 1998.
19. M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Int. Conf. on Softw. Eng.*, 1999.
20. D.R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Op. Syst. Design and Impl.*, pages 1–16, 2000.
21. J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *Trans. on Prog. Lang. and Syst.*, 3(9):319–349, 1987.
22. A. Finkel, B.Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Elec. Notes in Theor. Comp. Sci.*, 9, 1997.
23. Fast Library Identification and Recognition Technology, DataRescue sa/nv, Liège, Belgium, http://www.datarescue.com/idabase/flirt.htm.
24. B. Guo, M.J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D.I. August. Practical and accurate low-level pointer analysis. In *3nd Int. Symp. on Code Gen. and Opt.*, pages 291–302, 2005.
25. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Softw. Tools for Tech. Transfer*, 2(4), 2000.
26. J.L. Hennessy. Symbolic debugging of optimized code. *Trans. on Prog. Lang. and Syst.*, 4(3):323–344, 1982.
27. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Princ. of Prog. Lang.*, pages 58–70, 2002.
28. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Trans. on Prog. Lang. and Syst.*, 12(1):26–60, January 1990.
29. M. Howard. Some bad news and some good news. October 2002. MSDN, Microsoft Corp., http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure10102002.asp.
30. IDAPro disassembler, http://www.datarescue.com/idabase/.
31. N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for weighted pushdown systems, 2004. http://www.cs.wisc.edu/wpis/wpds++/.

32. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *Computer Aided Verif.*, 2005.
33. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *European Symp. on Programming*, 2005.
34. G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Princ. of Prog. Lang.*, pages 119–132, 1999.
35. T. Reps and G. Rosay. Precise interprocedural chopping. In *Found. of Softw. Eng.*, 1995.
36. T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Static Analysis Symp.*, 2003.
37. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. of Comp. Prog.* To appear.
38. S. Schwoon. Moped system. http://www.fmi.uni-stuttgart.de/szs/tools/moped/.
39. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical Univ. of Munich, Munich, Germany, July 2002.
40. D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Dist. Syst. Security*, February 2000.
41. D.W. Wall. Systems for late code modification. In R. Giegerich and S.L. Graham, editors, *Code Generation – Concepts, Tools, Techniques*, pages 275–293. Springer-Verlag, 1992.
42. R.P. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Prog. Lang. Design and Impl.*, pages 1–12, 1995.
43. P.T. Zellweger. *Interactive Source-Level Debugging of Optimized Programs*. PhD thesis, Univ. of California, Berkeley, 1984.

Calculating Polynomial Runtime Properties

Hugh Anderson, Siau-Cheng Khoo, Stefan Andrei, and Beatrice Luca

Department of Computer Science, School of Computing,
National University of Singapore
`{hugh, khoosc, andrei, lucabeat}@comp.nus.edu.sg`

Abstract. Affine size-change analysis has been used for termination analysis of eager functional programming languages. The same style of analysis is also capable of compactly recording and calculating other properties of programs, including their runtime, maximum stack depth, and (relative) path time costs. In this paper we show how precise (not just big-$\mathcal{O}$) polynomial bounds on such costs may be calculated on programs, by a characterization as a problem in quantifier elimination. The technique is decidable, and complete for a class of size-change terminating programs with limited-degree polynomial costs. An extension to the technique allows the calculation of some classes of exponential-cost programs. We demonstrate the new technique by recording the calculation in numbers-of-function (or procedure) calls for a simple functional definition language, but it can also be applied to imperative languages. The technique is automated within the `reduce` computer algebra system.

1 Introduction

Polynomial runtime properties are considered essential in many applications. The ability to calculate such properties statically and precisely will contribute significantly to the analysis of complex systems. In real-time systems, the time-cost of a function or procedure may be critical for the correct operation of a system, and may need to be calculated for validation of the correct operation of the system. For example, a device-driver may need to respond to some device state change within a specified amount of time.

In other applications, the maximum stack usage may also be critical in (for example) embedded systems. In these systems, the memory available to a process may have severe limitations, and if these limits are exceeded the behaviour of the embedded system may be unpredictable. An analysis which identifies the maximum depth of nesting of function or procedure calls can solve this problem, as the system developer can make just this amount of stack available.

A third motivation for calculating polynomial runtime properties is to calculate more precise relative costs of the individual calls. For example in a flow analysis of a program we may be interested in which calls are used most often, with a view to restructuring a program for efficiency. In this scenario, the relative costs between the individual calls is of interest. In the gcc compiler, a static branch predictor [3] uses heuristics to restructure the program code, optimizing the location of code for a branch more likely to occur. The approach described here can calculate more precise relative costs to improve these heuristics.

In this paper we explore the automatic calculation of each of these costs through static analysis of the source of programs which are known to be affine size-change terminating [2,17], where the focus is on recording parameter size-changes only. The overall approach has three steps:

1. Assume a (degree k) polynomial upper bound related to the runtime or space cost. The polynomial variables are the parameter *sizes*.
2. Derive from the source a set of equations constrained by this upper bound.
3. Solve the equations to derive a symbolic representation of the precise cost.

If the equations reduce to a vacuous result, then the original assumption of the degree of the polynomial must have been incorrect, and we repeat the process with a degree $k + 1$ assumption. This technique is surprisingly useful, and it is possible to derive precise runtime bounds on non-trivial programs.

We can also calculate the time or space costs for a subclass of exponential costs, in particular those of the form $\phi_1 \cdot K^{\phi_2} + \phi_3$ where ϕ_1, ϕ_2 and ϕ_3 are each a limited-degree polynomial in the parameter sizes, and $K \in \Re$ is a constant.

In the approach presented here, we measure runtime in terms of the number of calls to each procedure in a simple definition language. This is an appropriate measure, as the language does not support iteration constructs, and recursive application of procedures is the only way to construct iteration. Note that this approach does not restrict the applicability of the technique. Any iteration construct can be expressed as a recursion with some simple source transformation.

In Section 2, we position our work in relation to other research. In Section 3, preliminary concepts and definitions are introduced. In Section 4, the framework used for constructing the equations is introduced, along with practical techniques that may be used to solve the equations. In Section 5, we show examples of relative time costs for compiler optimization, and calculation of stack depth. In Section 6, we use recurrence relations to indicate how to classify costs into polynomial or exponential forms. In Section 7, exponential cost calculations are explored, before concluding in Section 8.

2 Related Work

There has been some research into run-time analysis for functional programs. For example, [21] explores a technique to evaluate a program's execution costs through the construction of recurrences which compute the time-complexity of expressions in functional languages. It focuses on developing a calculus for costs, and does not provide automated calculations. In [11], Grobauer explores the use of recurrences to evaluate a DML program's execution costs. Our focus is more with decidability aspects and precise time-costs than either of these approaches.

An alternative approach is to limit the language in some way to ensure a certain run-time complexity. For example, in [12], Hofmann proposes a restricted type system which ensures that all definable functions may be computed in polynomial time. The system uses inductive datatypes and recursion operators. In our work, we generate time and stack costs of arbitrary functions or procedures, through analysis of the derived size-change information.

The works [1,4,5] are clearly related to our work, although they focus on the polytime (i.e. PTIME) functions. The functions with time costs expressible as a polynomial are obviously included in the polytime functions, but our approach cannot handle (for example) `max`, and so a function $g(x, y)$ with a runtime of $\max(x, y)$, although computable in polytime is not calculated with our method. As well, a function $h(x, y)$ with a runtime of 2^{x+y}, is not computable in polytime, but is calculated with our method. The class we consider is thus not the same as *Polytime*.

Gómez and Liu [10] devise an automatic time-bound analysis for a higher-order language by translating programs into time-bound functions which compute time value, representing a bound to the actual time run by the program with specific program inputs. Time is not represented symbolically, but functionally. In a similar vein, a system to derive upper time-cost bounds of a first order subset of LISP is described in [19]. This approach is given in an abstract interpretation setting, and derives a program to compute the time bound function, in two phases, first deriving a step-counting version of the program, and then expressing the time bound function as an abstract interpretation of the step-counting program. As mentioned before we focus in this paper on precise calculations of time costs, and derive a closed form (see Definition 2). By contrast, the time bound function must be executed to discover the time cost, and itself may not terminate.

A compact summary of a general technique for the calculation of time and space efficiency is found in the book [20] by Van Roy and Haridi, where recurrence relations are used to model the costs of the language elements of the programming language. There is unfortunately no general solution for an arbitrary set of recurrence relations, and in practice components of the costs are ignored, capturing at each stage only the most costly recurrence, and leading to big-$\mathcal{O}$ analysis.

Our paper improves the technique for a specific class of functions, calculating more precise bounds than those derived from big-$\mathcal{O}$ analysis. By exploiting a-priori knowledge that a particular function terminates, and that the (polynomial) degree of the particular function is bounded, we can derive a formula which gives precisely the time or space cost of the program.

3 Preliminaries

The language in Table 1 is in some sense an *abstract* language, omitting any parts not relevant to the runtime. In addition, the expressions are given as if they were all integer values, when in fact they refer to expressions based on the *size* of the data types of the language. Size-change analysis operates over well-founded data structures such as lists or trees, and so a list may be represented here by a *size* integer representing the length of the list, and list concatenation represented by addition of the size values. The ˜ operation is a catch-all for any

Table 1. The language syntax

v	$\in$ `Var`	⟨Variables⟩
f, g, h	$\in$ `PName`	⟨Procedure names⟩
n	$\in \mathbb{Z}$	⟨Integer constants⟩
β	$\in$ `Guard`	⟨Boolean expressions⟩
	$\beta ::= \delta \mid \neg\beta \mid \beta_1 \vee \beta_2 \mid \beta_1 \wedge \beta_2$	
	$\delta ::= \text{True} \mid \text{False} \mid e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid e_1 \leq e_2 \mid e_1 \geq e_2$	
e	$\in$ `AExp`	⟨Expressions⟩
	$e ::= n \mid v \mid n \star e \mid e_1 + e_2 \mid -e$	
s	$\in$ `Stat`	⟨Statements⟩
	$s ::=$ `if` β `then` s_1 `else` s_2 `|` s_1`;`s_2 `|` $f(e_1, \ldots, e_n)$ `|` `~`	
d	$\in$ `Decl`	⟨Definitions⟩
	$d ::= f\ (x_1, \ldots, x_n) = s;$	

program operations that do not result in a function call. Finally, the language only admits affine relations between the program variables and expressions.

3.1 Runtime Analysis

In the process of performing size-change termination analysis described in [18], arbitrary sets of functions are processed, constructing a finite set of idempotent SCGs (Size-Change Graphs). These SCGs characterize the function, and detail all the ways in which a particular function entry point may be re-entered. In the following description, the functions are all derived from an affine SCT (Size-Change Termination) analysis [2,17], and hence are known to terminate. A subclass of these functions in which argument size-changes are linear, termed LA-SCT (Linear-affine SCT programs) define the class of programs analysed here. Limiting our analysis to this class of functions is not a severe restriction, as most useful size-change parameter changes would be linear.

We begin by formally defining the runtime of such functions. The term $\bar{y}$ refers to the vector $(y_1, \ldots, y_n)$. For the sake of notational brevity, we use a *contextual notation* to represent an expression containing *at most* one function call. For an expression containing a function call $f(\bar{y})$, the corresponding contextual notation is $\mathcal{C}[f(\bar{y})]$. For an expression containing no call, the corresponding contextual notation is $\mathcal{C}[]$.

Definition 1. *Given an LA-SCT program p with program parameters $\bar{x}$ and body e_p and input arguments $\bar{n}$, the runtime of p, $B(p)[\bar{n}/\bar{x}]$, is defined by the runtime of e_p inductively as follows:*

$$\begin{aligned}
&B(s_1; s_2)[\bar{n}/\bar{x}] && \stackrel{\text{def}}{=} B(s_1)[\bar{n}/\bar{x}] + B(s_2)[\bar{n}/\bar{x}] \\
&B(\texttt{if}\, g\, \texttt{then}\, s_1\, \texttt{else}\, s_2)[\bar{n}/\bar{x}] && \stackrel{\text{def}}{=} \text{if } g[\bar{n}/\bar{x}] \text{ then } B(s_1)[\bar{n}/\bar{x}] \text{ else } B(s_2)[\bar{n}/\bar{x}] \\
&B(\mathcal{C}[])[\bar{n}/\bar{x}] && \stackrel{\text{def}}{=} 0 \\
&B(\mathcal{C}[f(\bar{m})])[\bar{n}/\bar{x}] && \stackrel{\text{def}}{=} B(e_f)[\bar{m}/\bar{y}] + 1 \qquad (\text{where } e_f \text{ is the body of } f(\bar{y}))
\end{aligned}$$

In practical terms, this indicates that we are *counting* function calls as a measure of runtime. Such calls are the only difficult part of a runtime calculation, as other program constructs add *constant* time delays. To clarify this presentation, we choose to limit the definition to the cost analysis of function calls only.

In the case of a function $f(\bar{x})$ containing only a direct call $h(\bar{y})$, where $\bar{y} = \bar{x}[\psi]$, $[\psi] = [y_1 \mapsto \delta_1(x_1, x_2, \ldots), y_2 \mapsto \delta_2(x_1, x_2, \ldots), \ldots]$ and δ_1, δ_2 represent affine terms in the input parameters, we have:

$$B(f(\bar{x})) = B(h(\bar{x}[\psi])) + 1$$

We are primarily interested in runtimes that can be expressed as a polynomial in the parameter variables. We differentiate between an assumption $A_k(p)$ of the runtime of a program p, and the actual runtime $B_k(p)$.

Definition 2. *The degree-k polynomial runtime $B_k(p)$ of an LA-SCT program p with m parameters $\overline{x} = x_1, \ldots, x_m$ is a multivariate degree-k polynomial expression: $B_k(p) \stackrel{\text{def}}{=} c_1x_1^k + c_2x_2^k + \ldots + c_mx_m^k + c_{m+1}x_1^{k-1}x_2 + \ldots + c_n$, where $c_1 \ldots c_n \in \mathbb{Q}$, and $B_k(p)$ is the runtime of the program.*

An example of such a runtime for a program $p(x, y)$ is $B_2(p) = x + \frac{1}{2}y^2 + \frac{3}{2}y$.

Definition 3. *An assumption $A_k(p)$ of a polynomial runtime of an LA-SCT program p with m parameters $\overline{x} = x_1, \ldots, x_m$ is a multivariate polynomial expression: $A_k(p) \stackrel{\text{def}}{=} c_1x_1^k + c_2x_2^k + \ldots + c_mx_m^k + c_{m+1}x_1^{k-1}x_2 + \ldots + c_n$, where $c_1 \ldots c_n$ are unknown. $A_k(p)$ contains all possible terms of degree at most k formed by the product of parameters of p. Note that in this presentation, we search for an assignment $[\theta]$ to the constants $c_1 \ldots c_n$ such that $B_k(p) = A_k(p)[\theta]$.*

If a program p had two parameters x and y, then

$$\begin{aligned}
A_1(p) &= c_1x + c_2y + c_3\\
A_2(p) &= c_1x^2 + c_2y^2 + c_3xy + c_4x + c_5y + c_6\\
A_3(p) &= c_1x^3 + c_2y^3 + c_3x^2y + c_4xy^2 + c_5x^2 + c_6y^2 + c_7xy + c_8x + c_9y + c_{10}
\end{aligned}$$

In this presentation, we capture runtime behaviour by deriving sets of equations of the form $A_k(p(\bar{x})) = \sum(A_k(f_i(\bar{x}[\psi_i])) + 1)$ for each of the sets of calls f_i which are calls isolated and identified by the same guard. The substitution ψ_i relates the values of the input parameters to p to the values of the input parameters on the call f_i. Note that with this formulation, each substitution is linear, and thus cannot change the degree of the equation.

4 Characterization as a Quantifier-Elimination Problem

The sets of assumptions and runtimes presented in the previous section are universally quantified over the parameter variables, and this leads to the idea of formulating this problem as a QE (quantifier-elimination) one. Consider the following program p_1 operating over the naturals with parameters $x, y \in \mathbb{N}$:

```
p₁(x,y) = if ($x = 0 \wedge y \geq 1$) then $p_{1a}(y, y-1)$
          else if ($x \geq 1$)     then $p_{1b}(x-1, y)$
          else  ~ ;                 // ... exit ...
```

We can represent the runtime properties for each path through the program p_1 with the three equations:

$$\begin{aligned} A_2(p_1)[x \mapsto y, y \mapsto y-1] - A_2(p_1) + 1 &= 0 \\ A_2(p_1)[x \mapsto x-1] - A_2(p_1) + 1 &= 0 \\ A_2(p_1) &= 0 \end{aligned}$$

which reduce to:

$$\begin{aligned} -c_1x^2 + (c_1+c_3)y^2 - c_3xy - c_4x + (c_4 - c_3 - 2c_2)y + c_2 - c_5 + 1 &= 0 \\ c_1 - 2c_1x - c_3y - c_4 + 1 &= 0 \\ c_1x^2 + c_2y^2 + c_3xy + c_4x + c_5y + c_6 &= 0 \end{aligned}$$

We wish to find suitable values for the (real-valued) coefficients $c_1 \ldots c_6$. That is, we want to *eliminate the universally quantified elements* of the equalities.

There are several advantages of this QE formulation of the problem. Firstly, there is an automatic technique for solving sets of polynomial equalities and inequalities of this form, developed by Alfred Tarski in the 1930's, but first fully described in 1951 [23]. Tarski gives an inefficient decision procedure for a theory of elementary algebra of real numbers. Quantifier elimination is part of this theory, and after eliminating the quantified variables x and y in the above expressions, what remains are constraints over the values of the coefficients.

Secondly, precise analysis may be performed by including in the *guards* for each of the paths. For example, we can express our QE problem as the single formula[1]:

$$\begin{aligned} \forall x, y : \quad & \begin{pmatrix} x = 0 \\ \wedge\ y \geq 1 \end{pmatrix} \Rightarrow A_2(p_1)[x \mapsto y, y \mapsto y-1] - A_2(p_1) + 1 = 0 \\ \wedge \quad & (x \geq 1) \Rightarrow A_2(p_1)[x \mapsto x-1] - A_2(p_1) + 1 = 0 \\ \wedge \quad & \begin{pmatrix} x = 0 \\ \wedge\ y = 0 \end{pmatrix} \Rightarrow A_2(p_1) = 0 \end{aligned}$$

In [15], the author clearly shows how quantifier elimination may be used to generate program invariants using either a theory of Presburger arithmetic, a theory involving parametric Gröbner bases, or Tarski's theory of real closed fields. This last theory is the most expressive, and a claim is made that the approach is more widely applicable, and generates stronger invariants than the Gröbner basis approach in [22]. Our construction is different, and in a different field (runtime costs rather than program invariants).

4.1 Quantifier Elimination

In 1973, Tarski's method was improved dramatically by the technique of Cylindrical Algebraic Decomposition (CAD) described in [7,6], and leading to a quantifier free formula for a first order theory of real closed fields. In this theory, atomic formulæ may be of the form $\phi_1 = \phi_2$ or $\phi_1 > \phi_2$, where ϕ_1 and ϕ_2 are arbitrary polynomials with integer coefficients. They may be combined with the boolean connectives $\Rightarrow$, $\wedge$, $\vee$ and $\neg$, and variables may be quantified ($\forall$ and $\exists$).

[1] This derivation is automatic, and explained in an expanded version of this paper.

Definition 4. *A Tarski formula T is any valid sentence in the first order theory of real closed fields. Note that quantifier elimination is decidable in this theory.*

Our approach is to construct a particular subset of Tarski formulæ, $T[A(p)]$, where $A(p)$ is an assumption (defined before). This subset is of the form

$$T[A(p)] = \begin{pmatrix} \forall \overline{x}, \overline{y}, \ldots & g_1 \Rightarrow F_1 \\ & \wedge\ g_2 \Rightarrow F_2 \\ & \wedge \ldots\ldots\ldots \end{pmatrix}$$

where $g_1, g_2, \ldots$ identify different paths from $p(\overline{a})$ to enclosed function calls $f_i(\overline{b})^2$. $F_1, F_2, \ldots$ are formulæ derived from the program p source such that

$$\forall \overline{x} : g_j \Rightarrow (F_j \Leftrightarrow (A_k(p(\bar{x})) = \sum_i A_k(f_i(\bar{x}[\psi_i])) + 1))$$

Inference rules can be used to automatically generate these "Tarski" formulæ from an arbitrary input program. For example, for the program p_1:

$$A_2(p_1) \;=\; \begin{pmatrix} & (x = 0 \wedge y = 0) : & 0 \\ \wedge & (x = 0 \wedge y \geq 1) : & A_2(p_1)[x \mapsto y, y \mapsto y - 1] + 1 \\ \wedge & (x \geq 1) : & A_2(p_1)[x \mapsto x - 1] + 1 \end{pmatrix}$$

and the equation $T[A_2(p_1)]$ derived is thus:

$$T[A_2(p_1)] = \begin{pmatrix} \forall x, y : & \begin{pmatrix} x = 0 \\ \wedge\ y = 0 \end{pmatrix} \Rightarrow & A_2(p_1) = 0 \\ \wedge & \begin{pmatrix} x = 0 \\ \wedge\ y \geq 1 \end{pmatrix} \Rightarrow & A_2(p_1)[x \mapsto y, y \mapsto y - 1] - A_2(p_1) + 1 = 0 \\ \wedge & (x \geq 1) \Rightarrow & A_2(p_1)[x \mapsto x - 1] - A_2(p_1) + 1 = 0 \end{pmatrix}$$

and our task now is to reduce this to an expression without x and y, and then find any example of $c_1 \ldots c_6$ satisfying the resultant expression.

The following theorem asserts that the solution of the formula $T[A_k(p)]$ correctly represents the runtime $B_k(p)$ of any LA-SCT program p with a degree-k polynomial runtime.

Theorem 1. *If $B_k(p)$ is the degree-k polynomial runtime of affine SCT program p with parameters $\bar{x}$, and $A_k(p)$ is a degree-k polynomial assumption of the runtime of LA-SCT program p, and $[\theta]$ is the assignment derived from $T[A_k(p)]$, then*

$$\forall \bar{n} : \quad A_k(p)[\theta][\bar{n}/\bar{x}] \equiv B_k(p)[\bar{n}/\bar{x}]$$

Proof. By structural induction over the form of the definition for $B_k(p)[\bar{n}/\bar{x}]$.

[2] Note that they must cover the parameter space of interest and be distinct.

4.2 Tool Support

There are a range of tools capable of solving this sort of reduction. The tool `QEP-CAD` [9], and the `redlog` package [8] in the computer algebra system `reduce`, may be used to eliminate quantifiers giving completely automatic results. Consider the `redlog` commands that *specify* the runtime for program p_1:

```
1: A2p1  := c1*x^2+c2*y^2+c3*x*y+c4*x+c5*y+c6;
2: path1 := sub(x=y,y=y-1,A2p1)-A2p1+1;
3: path2 := sub(x=x-1,A2p1)-A2p1+1;
```

In line 1 of the above sequence, we define the `A2p1` assumption of the runtime bounds B_2 of the program. In lines 2 and 3, $A_2(p_1)[x \mapsto y, y \mapsto y-1] - A_2(p_1)+1$ and $A_2(p_1)[x \mapsto x-1] - A_2(p_1) + 1$ (The `sub` command in `reduce` performs a series of substitutions in the expression `A2p1`). The following sequence shows the `redlog` commands to *solve* the problem:

```
4: TA2p1 := rlqea ex({c1,c2,c3,c4,c5,c6},
                      rlqe all({x,y},
                              ((x=0 and y=0)  impl A2p1=0) and
                              ((x=0 and y>=1) impl path1=0) and
                              ((x>=1)         impl path2=0)));
5: B2p1  := sub( part(part(TA2p1,1),2),A2p1);
```

In line 4 of the above sequence, the inner `rlqe` function performs quantifier elimination on the equation $T[A_2(p_1)]$, returning the following relations:

$$c_4 = 1 \wedge 2c_2 - c_4 = 0 \wedge c_2 - c_5 = -1 \wedge c_1, c_3, c_6 = 0$$

In this example, $c_1 \ldots c_6$ are uniquely determined, and can be found easily with a few simple reductions, but in the general case, the constraints over the constants may lead to many solutions. The `redlog` package can also be used to find an instance of a solution to an existentially quantified expression, and hence the outer `rlqea` function above, which returns an instance of a solution to the above relations existentially quantified over $c_1 \ldots c_6$: $\exists c_1 \ldots c_6 \, : \, T[A_2(p_1)]$. The solution returned by `redlog` is:

```
TA2p1 := {{true,{c1=0, c2=1/2, c3=0, c4=1, c5=3/2, c6=0}}}
```

Finally, in line 5, we substitute the solution instance back in the original assumption $A_2(p_1) = c_1x^2 + c_2y^2 + c_3xy + c_4x + c_5y + c_6$, giving

$$\begin{aligned} B_2(p_1) &= A_2(p_1)[c_1 \mapsto 0, c_2 \mapsto \frac{1}{2}, c_3 \mapsto 0, c_4 \mapsto 1, c_5 \mapsto \frac{3}{2}, c_6 \mapsto 0] \\ &= x + \frac{1}{2}y^2 + \frac{3}{2}y \end{aligned}$$

We might consider a constraint programming based solution to these sort of problems, and there are constraint solving systems, for example `RISC-CLP(Real)` [13], which use (internally) CAD quantifier elimination to solve polynomial constraints. However here we prefer to restrict ourselves to just the underlying techniques, and not clutter up the discussion with other properties of constraint solving systems.

5 Calculating Other Program Costs

So far we have limited the presentation to examples which calculate polynomial runtimes for programs. However, the technique is also useful for deriving other invariant properties of programs, such as stack depth and *relative* time costs.

5.1 Stack Depth Calculation

Consider program p_2:

```
    p2(x,y) = if (x = 0 ∧ y ≥ 1) then
                  p2a(y, y − 1);
                  p2b(0, y − 1)
              else if (x ≥ 1) then  p2c(x − 1, y)
              else ~ ;              // ... exit ...
```

Note that in this program, we have the sequential composition of two function calls, and this program has an exponential runtime cost. The depth D of our class of programs is calculated in precisely the same way as the runtime B, with only a minor change. In the event of sequential composition, we record not the sum of the two functions composed, but the maximum value of the two functions. This corresponds with a Tarski formula for a polynomial solution like this:

$$\begin{pmatrix} \forall x, y: & (x = 0 \wedge y \geq 1 \wedge D[\psi_{2a}] \geq D[\psi_{2b}]) \Rightarrow (D[\psi_{2a}] - D + 1 = 0) \\ & \wedge\ (x = 0 \wedge y \geq 1 \wedge D[\psi_{2a}] < D[\psi_{2b}]) \Rightarrow (D[\psi_{2b}] - D + 1 = 0) \\ & \wedge \qquad\qquad\qquad\qquad\qquad\qquad (x \geq 1) \Rightarrow (D[\psi_{2c}] - D + 1 = 0) \end{pmatrix}$$

Given the formula, `redlog` finds the stack *depth* cost: $D(p_2) = x + \frac{1}{2}y^2 + \frac{3}{2}y$.

5.2 Relative Runtime Costs

The third motivation for this approach was to derive relative costs for the different possible paths through a program. For example in program p_1, which function is called more often, and what are the relative costs for each call? This could be used in compiler optimization, improving the efficiency of the code by re-ordering and placing more commonly used functions nearby.

The same approach may be used, calculating B for each path. The equation $T[A(p_{1a})]$ for the program choosing the first function call may be written as:

$$T[A(p_{1a})] = \begin{pmatrix} \forall x, y: & \begin{pmatrix} x = 0 \\ \wedge\ y = 0 \end{pmatrix} \Rightarrow & A_2(p_1) = 0 \\ \\ \wedge & \begin{pmatrix} x = 0 \\ \wedge\ y \geq 1 \end{pmatrix} \Rightarrow & A_2(p_1)[x \mapsto y, y \mapsto y-1] - A_2(p_1) + 1 = 0 \\ \\ \wedge & (x \geq 1) \Rightarrow & A_2(p_1)[x \mapsto x-1] - A_2(p_1) = 0 \end{pmatrix}$$

$$\Rightarrow\ B_2(p_{1a}) = y$$

The equation $T[A(p_{1b})]$ for the program choosing the second function call may be written as:

$$T[A(p_{1b})] = \left(\begin{array}{lll} \forall x, y : & \left(\begin{array}{c} x = 0 \\ \wedge\ y = 0 \end{array}\right) \Rightarrow & A_2(p_1) = 0 \\ \\ \wedge & \left(\begin{array}{c} x = 0 \\ \wedge\ y \geq 1 \end{array}\right) \Rightarrow & A_2(p_1)[x \mapsto y, y \mapsto y-1] - A_2(p_1) = 0 \\ \\ \wedge & \quad (x \geq 1) \Rightarrow & A_2(p_1)[x \mapsto x-1] - A_2(p_1) + 1 = 0 \end{array} \right)$$

$$\Rightarrow \ B_2(p_{1b}) \ = \ x + \frac{1}{2}(y^2 + y)$$

The sum of $B_2(p_{1a})$ and $B_2(p_{1b})$ is exactly $B_2(p_1)$ for the whole program.

6 Towards a Classification of Program Costs

The presentation so far has concentrated on LA-SCT programs with costs that may be expressed as polynomials over the program variables. However many such programs have costs that are exponential rather than polynomial. For example, the following program:

```
$p_3$( $x, y, n$ ) = if ($x \neq 0 \wedge n \geq 1$)         then $p_{3a}$( $x-1, y, n$ )
                     else if ($x = 0 \wedge n > 1$)  then $p_{3b}$( $2y+n, 2y, n-1$ )
                     else  ˜ ;                  // ... exit ...
```

This program has a runtime of $B(p_3) = y2^n + \frac{1}{2}n^2 + \frac{3}{2}n + x - 2y - 2$, not immediately apparent by observation. The technique such as just described relies on repeatedly trying ever higher degree polynomial time costs, and would never discover this runtime. For example, if we started assuming the program was polynomial, the algorithm indicates that we should try a degree-2 assumption, followed by a degree-3 assumption and so on. There is no indication as to when we should give up.

We have an approach to solving programs of this form, but it requires us to find some way of classifying program costs into either polynomial or exponential. In this section we present a characterization of the problem as a recurrence, explaining the choice of the particular class of exponential cost programs that can be solved. Towards this, we consider a *flattened* version of the original program source, in which an arbitrary collection of functions is flattened into a single function which calls itself. This new flattened source can be easily characterized as a recurrence relation, and the solutions to the recurrence relations give indications of the maximum polynomial degree.

A flattened version of an arbitrary program is easily derived in the absence of mutual recursion. However, in the case of mutually recursive functions, it is not as clear how a program may be transformed. The papers [24,16] contain necessary and sufficient conditions to transform all mutual recursion to direct or self-recursion. Supposing that our programs are transformed into equivalent programs which are using only self-recursion, we can define a *self-recursive normal form* over a representation of the state of the program variables at any time. Consider an m-dimensional array a, indexed by the values of parameters $n_1 \dots n_m$ to the self-recursive program $p(n_1 \dots n_m)$:

Definition 5. *The array $a_{n_1,\dots,n_m}$ is in linear self-recursive normal form iff it is defined as:*

$$a_{n_1,\dots,n_m} = a_{f_1(n_1,\dots,n_m),\dots,f_m(n_1,\dots,n_m)} + g(n_1, ..., n_m) \tag{1}$$

where $f_i(n_1, ..., n_m) = k_{i,1} \cdot n_1 + \dots + k_{i,m} \cdot n_m + k_{i,m+1}$, $\forall k_{i,j} \in \Re$, $\forall i \in \{1, \dots, m\}$, $\forall j \in \{1, \dots, m+1\}$, and $g(n_1, ..., n_m) = k_1 \cdot n_1 + \dots + k_m \cdot n_m + k_{m+1}$.

The above recurrence (1) is supposed to iterate for an arbitrary finite number of times, say ℓ. We shall explore the expression obtained from (1) after applying the substitution $n_i \to f_i(n_1, \dots, n_m)$, $\forall i \in \{1, \dots, m\}$ for ℓ times.

Theorem 2. *All linear self-recursive normal forms have a solution.*

Proof. (By construction). Denoting by $\overline{n}$ the vector $(n_1, \dots, n_m)$, the first iteration of (1) leads to:

$$a_{f_1(\overline{n}),\dots,f_m(\overline{n})} = a_{f_1(f_1(\overline{n}),\dots,f_m(\overline{n})),\dots,f_m(f_1(\overline{n}),\dots,f_m(\overline{n}))} + g(f_1(\overline{n}), ..., f_m(\overline{n})) \tag{2}$$

In order to write this more compactly, let us inductively define the notations

$$\left\langle f_{1,m}^{(1)}(\overline{n}) \right\rangle \overset{\text{def}}{=} (f_1(\overline{n}), ..., f_m(\overline{n}))$$
$$\left\langle f_{1,m}^{(\ell)}(\overline{n}) \right\rangle \overset{\text{def}}{=} \left(f_1 \left(\left\langle f_{1,m}^{(\ell-1)}(\overline{n}) \right\rangle \right), \dots, f_m \left(\left\langle f_{1,m}^{(\ell-1)}(\overline{n}) \right\rangle \right) \right) \quad \text{for } \ell \geq 2$$

where $\left\langle f_{1,m}^{(1)}(\overline{n}) \right\rangle$ is a compressed form of $\langle f_{1,m} \circ ... \circ f_{1,m}(\overline{n}) \rangle$, and "$\circ$" stands for the function composition. The recurrence (2) can then be re-written as:

$$a_{\left\langle f_{1,m}^{(1)}(\overline{n}) \right\rangle} = a_{\left\langle f_{1,m}^{(2)}(\overline{n}) \right\rangle} + g \left(\left\langle f_{1,m}^{(1)}(\overline{n}) \right\rangle \right) \tag{2a}$$

The given substitution can be further applied $\ell - 1$ times, to obtain:

$$a_{\left\langle f_{1,m}^{(\ell-1)}(\overline{n}) \right\rangle} = a_{\left\langle f_{1,m}^{(\ell)}(\overline{n}) \right\rangle} + g \left(\left\langle f_{1,m}^{(\ell-1)}(\overline{n}) \right\rangle \right) \tag{ℓ}$$

equation By combining the recurrences $(1) \dots (\ell)$, we obtain an expression for $a_{\overline{n}}$:

$$a_{\overline{n}} = a_{\left\langle f_{1,m}^{(\ell)}(\overline{n}) \right\rangle} + g \left(\left\langle f_{1,m}^{(1)}(\overline{n}) \right\rangle \right) + \dots + g \left(\left\langle f_{1,m}^{(\ell-1)}(\overline{n}) \right\rangle \right) \tag{I}$$

By replacing $f_i(n_1, ..., n_m)$ with $k_{i,1} \cdot n_1 + \dots + k_{i,m} \cdot n_m + k_{i,m+1}$, $\forall i \in \{1, \dots, m\}$, we get the general form: $\left\langle f_{1,m}^{(l)}(\overline{n}) \right\rangle = (E_{1,\ell}, \dots, E_{m,\ell})$, where $E_{i,l}$ is:

$$\sum_{i_l=1}^{m} \cdots \sum_{i_1=1}^{m} k_{i,i_1} \cdot \dots \cdot k_{i_{l-1},i_l} \cdot n_{i_l} + \sum_{i_{l-1}=1}^{m} \cdots \sum_{i_1=1}^{m} k_{i,i_1} \cdot \dots \cdot k_{i_{l-1},m+1} + \dots + \sum_{i_1=1}^{m} k_{i,i_1} \cdot \dots \cdot k_{i_1,m+1}$$

This is a solution for all recurrences of the self-recursive normal form defined before, confirming the completeness for this class of recursive programs. $\square$

For ease of presentation, and in order to see the complexity of $a_{\overline{n}}$ from (I), let us highlight only the last (dominant) term. It is:

$$g\left(\left\langle f_{1,m}^{(\ell-1)}(\overline{n})\right\rangle\right) = k_1 \cdot E_{1,l-1} + \ldots + k_m \cdot E_{m,l-1} + k_{m+1}$$

Looking at the general form of the dominant term, namely

$$k_i \cdot \sum_{i_{l-1}=1}^{m} \ldots \sum_{i_1=1}^{m} k_{i,i_1} \cdot \ldots \cdot k_{i_{\ell-1},m+1} + \ldots + \sum_{i_1=1}^{m} k_{i,i_1} \cdot \ldots \cdot k_{i_1,m+1}$$

we observe that very few cases correspond to a polynomial as an expression for $a_{\overline{n}}$. Because of the large number of coefficients in the expression of $a_{\overline{n}}$, it is almost impossible to provide a precise boundary between the cases when $a_{\overline{n}}$ is a polynomial and when it is an exponential. However, the formula does immediately give the following classifications:

1. if $k_i = 0$ for all $i \in \{1, \ldots, m\}$, then $a_{\overline{n}} = \ell \cdot k_{m+1}$ is a polynomial in ℓ of degree 1;
2. if $m = 1$ and $k_{1,1} = 1$ then $a_{\overline{n}}$ is a polynomial of degree 2;
3. if $m = 1$ and $k_{1,1} \neq 1$ then $a_{\overline{n}}$ is an exponential of base $k_{1,1}$.
4. if $\exists i \in \{1, \ldots, m\}$ such that $k_i \neq 0$ and $\exists u, v \in \{1, \ldots, m\}$ such that $k_{u,v} \notin \{0, 1\}$ then $a_{\overline{n}}$ contains at least one exponential of base $k_{u,v}$.

The fourth classification above covers a considerable number of situations when $a_{\overline{n}}$ is an exponential. A useful slight generalization of recurrence (1) can be done by taking g as a non-linear polynomial. It is easy to see that if $m = 1$, and $k_{1,1} = 1$, then for a polynomial g of degree k, the solution of $a_{\overline{n}}$ is a polynomial of degree $k+1$. We have enlarged the class of self-recursive normal form equations.

The automation of the classification process is possible through procedure inlining described in [24,16], and the use of algebraic simplification tools. After inlining, we check if the form of the recursive function maps onto the linear self-recursive normal form.

6.1 A Case-Study

Let us take a useful example which corresponds to particular values for m, followed by a practical application of its use in computing the runtime of a given program. When trying to compute the runtime cost of p_3, we get the following identities, formed by a guard and a recurrence relation:

$$\begin{array}{lll} x \neq 0 \wedge n \geq 1 & \text{implies} & B(x, y, n) = B(x-1, y, n) + 1 \\ x = 0 \wedge n > 1 & \text{implies} & B(x, y, n) = B(2y+n, 2y, n-1) + 1 \end{array}$$

By inspection of the first identity, and by iterating $x \to x - 1$ for x times, we get $B(x, y, n) = B(0, y, n) + x$. By applying the second identity, we have $B(0, y, n) = B(2y + n, 2y, n - 1) + 1 = B(0, 2y, n - 1) + 2y + n + 1$. We rewrite this latter identity, omitting the first argument (without loss of generality), to the equivalent recurrence relation:

$$a_{y,n} = a_{2y,n-1} + 2y + n + 1$$

This is a particular instance of recurrence (1), where m is replaced by y, and $f(n,m) = 2m$, $g(n,m) = 2m + n + 1$. Since $k_1 = 2$, the solution of $a_{y,n}$ is an exponential (case 2(b)). This implies that our automated tool should be fed with an input having a generic form like $B(p_3) = \phi_1 \cdot K^{\phi_2} + \phi_3$, allowing for a runtime with quite a complex exponential form.

7 Exponential Cost Calculations

Having established a classification of program costs, we now revert to the original approach, where we assume an exponential runtime A for the program, initially for a base of K, and using polynomials of (say) degree 2. The assumed runtime is $A(p_3) = \phi_1 \cdot K^{\phi_2} + \phi_3$, where ϕ_1, ϕ_2 and ϕ_3 are three polynomials of degree 2. The three polynomials bear a peculiar relationship to each other due to the linearity of the parameter relationships. For example, for any single recursive call path, since the changes in the parameters are linear, then the runtime for this call path cannot be exponential. As a result of this, for any single recursive call path, $\phi_3[\psi] - \phi_3 + 1 = 0$, and the following relation holds:

$$\begin{array}{rl} & (\phi_1[\psi] = \phi_1 \quad \wedge \phi_2[\psi] = \phi_2) \\ \vee & (\phi_1[\psi] = K\phi_1 \wedge \phi_2[\psi] = \phi_2 - 1) \\ \vee & (K\phi_1[\psi] = \phi_1 \quad \wedge \phi_2[\psi] = \phi_2 + 1) \end{array}$$

This relationship between the polynomials may be exploited by constructing the equations in a similar form to the previous presentation, solving them, and finally deriving a sample solution. The `redlog` package is used to define

$$\begin{aligned} \phi_1 &= c_1x^2 + c_2y^2 + c_3n^2 + c_4xy + c_5xn + c_6yn + c_7x + c_8y + c_9n + c_{10} \\ \phi_2 &= c_{11}x^2 + c_{12}y^2 + c_{13}n^2 + c_{14}xy + c_{15}xn + c_{16}yn + c_{17}x + c_{18}y + c_{19}n \\ \phi_3 &= c_{21}x^2 + c_{22}y^2 + c_{23}n^2 + c_{24}xy + c_{25}xn + c_{26}yn + c_{27}x + c_{28}y + c_{29}n + c_{30} \\ A &= \phi_1 \cdot K^{\phi_2} + \phi_3 \end{aligned}$$

The substitutions $[\psi_{3a}] = [x \mapsto x-1]$ and $[\psi_{3b}] = [x \mapsto 2y+n, y \mapsto 2y, n \mapsto n-1]$ for the two paths are applied to ϕ_1, ϕ_2 and ϕ_3, yielding the primed polynomials, and the equation $T[A(p_3)]$ for program p_3 may be written as:

$$\left(\begin{array}{lcl} \forall x,y,n: \begin{pmatrix} x,y>0 \\ \wedge \quad n \geq 0 \end{pmatrix} & \Rightarrow & \left(\begin{array}{c} \phi_3[\psi_{3a}] - \phi_3 + 1 = 0 \\ \wedge \\ \left(\begin{array}{rl} & (\phi_1[\psi_{3a}] = \phi_1 \quad \wedge \phi_2[\psi_{3a}] = \phi_2) \\ \vee & (\phi_1[\psi_{3a}] = K\phi_1 \wedge \phi_2[\psi_{3a}] = \phi_2 - 1) \\ \vee & (K\phi_1[\psi_{3a}] = \phi_1 \quad \wedge \phi_2[\psi_{3a}] = \phi_2 + 1) \end{array}\right) \end{array}\right) \\ \qquad \wedge \quad \begin{pmatrix} x = 0 \\ \wedge\ y > 0 \\ \wedge\ n \geq 0 \end{pmatrix} & \Rightarrow & \left(\begin{array}{c} (\phi_3[\psi_{3b}] - \phi_3 + 1 = 0) \\ \wedge \\ \left(\begin{array}{rl} & (\phi_1[\psi_{3b}] = \phi_1 \quad \wedge \phi_2[\psi_{3b}] = \phi_2) \\ \vee & (\phi_1[\psi_{3b}] = K\phi_1 \wedge \phi_2[\psi_{3b}] = \phi_2 - 1) \\ \vee & (K\phi_1[\psi_{3b}] = \phi_1 \quad \wedge \phi_2[\psi_{3b}] = \phi_2 + 1) \end{array}\right) \end{array}\right) \end{array}\right)$$

$T[A(p_3)]$ is easily reduced by `redlog`, giving a family of solutions for the bounds:

$$A(p_3) = \alpha y 2^n + \frac{1}{2}n^2 + \frac{3}{2}n + x - 2y + c_{30}$$

where α indicates that any value here might be a solution, and c_{30} is unknown. To constrain the solution further, we add in boundary cases for the system, for example $A(p_3(0,1,1)) = 0$, $A(p_3(0,2,1)) = 0$, giving:

$$B(p_3) = y2^n + \frac{1}{2}n^2 + \frac{3}{2}n + x - 2y - 2$$

7.1 Another Example

Despite the simplicity of program p_2 introduced in subsection 5.1, a translation to a single-term recurrence is not obvious. The function would have to be flattened, generating extra guards and program parameters. However, the QE formulation is still automatic and simple, deriving the equation for the runtime cost $T[A(p_2)]$:

$$\left(\begin{array}{ll} \forall x, y: \quad (x = 0 \wedge y \geq 1) \Rightarrow (\phi_2[\psi_{2a}] + \phi_3[\psi_{2b}] - \phi_3 + 2 = 0) & \\ \wedge\ (x \geq 10 \wedge y \geq 0) \Rightarrow \left(\begin{array}{c} (\phi_3[\psi_{2c}] - \phi_3 + 1 = 0) \\ \wedge \\ \left(\begin{array}{ll} (\phi_1[\psi_{2c}] = \phi_1 & \wedge\ \phi_2[\psi_{2c}] = \phi_2) \\ \vee\ \ (\phi_1[\psi_{2c}] = K\phi_1 & \wedge\ \phi_2[\psi_{2c}] = \phi_2 - 1) \\ \vee\ (K\phi_1[\psi_{2c}] = \phi_1 & \wedge\ \phi_2[\psi_{2c}] = \phi_2 + 1) \end{array}\right) \end{array}\right) \end{array}\right)$$

Given two independent base cases, `redlog` immediately finds the runtime cost:

$$B(p_2) = 4 * 2^y + x - y - 4$$

We have found it relatively easy to automatically derive exponential runtimes for programs like these, with polynomials of small degree.

8 Conclusion

In this paper, we have shown a technique for calculating precise bounds on the runtime of a class of programs, which are known to terminate. The technique begins with an assumption of the form and degree of the runtime, and is complete in the sense that if the program p is LA-SCT, and if the runtime is of the form $B_k(p)$, then a solution will be found. The technique has application in the areas of precise runtime analysis, stack depth analysis, and in calculations of relative execution path time.

For practical use, the scalability of any approach such as this is an issue. The approach can be readily made more efficient by partitioning a program into the disjoint sets of functions, and then solving the sets independently. A known cost of a function may be substituted in other enclosing function calculations without having to reconstruct the graphs or equations. In this way there is a degree of compositionality that assists in addressing the scalability concern.

The class of functions that may be calculated by the method has not yet been completely identified. We can solve the functions with costs that are a precise polynomial, and those with a precise exponential cost of a particular form, and costs involving `min`. Work is continuing to try to clarify and categorize the class of functions that may be calculated by the method.

We have shown that the technique is safe and complete for the particular class of programs we have considered. In addition, we have presented an approach to classifying the costs into either polynomial or exponential time costs using a recurrence-relation complexity analysis. This outlined a particular form of exponential time-costs that can be relatively easily solved. In the case of the limited class of exponential time-costs, these solutions may still be expressed in terms of some unknowns, but these unknowns are resolved immediately by considering independent boundary cases for the function.

Acknowledgments. We thank Professor Neil Jones for initially proposing the form of this problem [14], Professor Luke Ong and the anonymous reviewers for their constructive comments, and the members of the PLS-II laboratory at NUS. This research is partially supported by NUS research grant R-252-000-138-112.

References

1. K. Aehlig and H. Schwichtenberg. A Syntactical Analysis of Non-Size-Increasing Polynomial Time Computation. *ACM Trans. Comput. Logic*, 3(3):383–401, 2002.
2. H. Anderson and S.C. Khoo. Affine-based Size-change Termination. In Atsushi Ohori, editor, *APLAS 03: Asian Symposium on Programming Languages and Systems*, pages 122–140, Beijing, 2003. Springer Verlag.
3. T. Ball and J.R. Larus. Branch Prediction For Free. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 300–313, 1993.
4. S. Bellantoni and S. Cook. A New Recursion-Theoretic Characterization of the Polytime Functions (Extended Abstract). In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 283–293, New York, NY, USA, 1992. ACM Press.
5. V.H. Caseiro. *Equations for Defining Poly-Time Functions.* PhD thesis, University of Oslo, 1997.
6. B.F. Caviness and J.R. Johnson (eds.). *Quantifier Elimination and Cylindrical Algebraic Decomposition.* Springer, 1998.
7. G.E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In H. Brakhage, editor, *Automata Theory and Formal Languages*, volume 33, pages 134–183, Berlin, 1975.
8. A. Dolzmann and T. Sturm. REDLOG: Computer algebra meets computer logic. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 31(2):2–9, 1997.
9. H. Hong et al. `http://www.cs.usna.edu/~qepcad/B/QEPCAD.html`.
10. G. Gómez and Y.A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *PEPM '02: Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 75–86, New York, NY, USA, 2002. ACM Press.
11. B. Grobauer. Cost Recurrences for DML Programs. In *International Conference on Functional Programming*, pages 253–264, 2001.
12. M. Hofmann. Linear Types and Non-Size-Increasing Polynomial Time Computation. In *Logic in Computer Science*, pages 464–473, 1999.
13. H. Hong. RISC-CLP(Real): Constraint Logic Programming over Real Numbers. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research.* MIT Press, 1993.

14. N. Jones. Private communication. June 2003.
15. D. Kapur. Automatically Generating Loop Invariants Using Quantifier Elimination. In *Proceedings of the 10th International Conference on Applications of Computer Algebra*. ACA and Lamar University, July 2004.
16. O. Kaser, C. R. Ramakrishnan, and S. Pawagi. On the Conversion of Indirect to Direct Recursion. *LOPLAS*, 2(1-4):151–164, 1993.
17. S.C. Khoo and H. Anderson. Bounded Size-Change Termination. Technical Report TRB6/05, National University of Singapore, June 2005.
18. C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. In *Conference Record of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM press, January 2001.
19. M. Rosendahl. Automatic Complexity Analysis. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 144–156, New York, NY, USA, 1989. ACM Press.
20. P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* The MIT Press, 2004.
21. D. Sands. Complexity Analysis for a Lazy Higher-Order Language. In *Proceedings of the Third European Symposium on Programming*, number 432 in LNCS, pages 361–376. Springer-Verlag, May 1990.
22. S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 318–329, 2004.
23. A. Tarski. In *A decision method for elementary algebra and geometry. Prepared for publication by J.C.C. Mac Kinsey.* Berkeley, 1951.
24. T. Yu and O. Kaser. A Note on "On the Conversion of Indirect to Direct Recursion". *ACM Trans. Program. Lang. Syst.*, 19(6):1085–1087, 1997.

A More Examples

The following extra examples illustrate a range of programs operating over naturals, with their automatically generated runtime costs.

Program 4:

```
p4(x,y)    = if (y ≤ 0) then  g(x,0)
             else             p4a(x+1, y−1);
g(x,y)     = if (x ≤ 0) then  ~              // ... exit ...
             else             g(x−1, y+1);
```

The solution returned by `redlog` is that $B_2(p_4) = x + \frac{1}{2}y^2 + \frac{3}{2}y + 1$.

Program 5:

```
p5(x,y)    = f(x,y,y+1);
f(x,y,z)   = if (y = z ∧ x > y − z)        then fa(x−1, y, z)
              else if (x = y − z ∧ y ≠ 0) then fb(−x, y−1, z)
              else if (x < y ∧ y ≠ 0)     then fc(x+1, y, z)
              else if (y < z ∧ x = y)     then fd(x, y, y)
              else  ~  ;              // ... exit ...
```

The solution returned by `redlog` is that $B_2(p_5) = y^2 + 3y - x + 1$.

Program 6:

```
p6( x, y, z )  = if  (x ≠ 0 ∧ z ≥ 1)       then p6a( x − 1, y + 1, z )
                 else if (x = 0 ∧ z ≥ 1) then p6b( 2y, 2y, z − 1 )
                 else   ˜                   // ... exit ...
```

The solution returned by `redlog` is that $B_2(p_6) = \frac{1}{6}((x+y)4^z + 6z + 2x - 4y - 6)$.

Program 7:

With a refinement of our approach not explored in this paper, we can also derive minimum values for costs, not just polynomial or restricted exponential costs. For example:

```
p7( x, y )     = if (x ≥ 1 ∧ y ≥ 1) then  p7a( x − 1, y − 1 )
                 else  ˜                  // ... exit ...
```

The solution returned by `redlog` is that $B_2(p_7) = \min(x, y)$.

Resource Bound Certification for a Tail-Recursive Virtual Machine*

Silvano Dal Zilio[1] and Régis Gascon[2]

[1] LIF, CNRS and Université de Provence, France
[2] LSV, CNRS and ENS Cachan, France

Abstract. We define a method to statically bound the size of values computed during the execution of a program as a function of the size of its parameters. More precisely, we consider bytecode programs that should be executed on a simple stack machine with support for algebraic data types, pattern-matching and tail-recursion. Our size verification method is expressed as a static analysis, performed at the level of the bytecode, that relies on machine-checkable certificates. We follow here the usual assumption that code and certificates may be forged and should be checked before execution.

Our approach extends a system of static analyses based on the notion of quasi-interpretations that has already been used to enforce resource bounds on first-order functional programs. This paper makes two additional contributions. First, we are able to check optimized programs, containing instructions for unconditional jumps and tail-recursive calls, and remove restrictions on the structure of the bytecode that was imposed in previous works. Second, we propose a direct algorithm that depends only on solving a set of arithmetical constraints.

1 Introduction

Bytecode programs are a form of intermediate code commonly used by language implementors when programs should be distributed and run on multiple platforms. Because of its advantages on performance and portability, many programming languages are actually compiled into bytecode. Java and Microsoft C# are representative examples, but bytecode compilers can also be found for less conventional languages, such as O'Caml, Perl or PHP. On the downside, bytecode typically stands at an abstraction level in between (high-level) source code and machine code: it is usually more compact and closer to the computer architecture than program code that is intended for "human consumption". Therefore, it is necessary to devise specific verification methods to guarantee properties at the bytecode level. For instance, to ensure the safety of executing newly loaded code, virtual machines generally rely on machine-checkable certificates that the program will comply with user-specific requirements. The interest of verification of such properties at the bytecode level is now well understood, see for example [14,17].

* This work was partly supported by ACI Sécurité Informatique, project CRISS.

As networked and mobile applications become more and more pervasive, and with the lack of third parties in control of trust management (like e.g. frameworks based on code signing), security appears as a major issue. Initial proposals for securing bytecode applications have focused on the integrity of the execution environment, such as the absence of memory faults and access violations. In this paper, we consider another important property, namely certifying bounds on the resources needed for the execution of the code. This problem naturally occurs when dealing with mobile code, for example to prevent denial of service attacks, in which the virtual machine is starved of memory by the execution of a malicious program. More precisely, we define a method to statically bound the size of values computed during the execution of a program. The size-bound obtained by this method is expressed as a function of the size of the parameters of the program (actually as a polynomial expression) and has several uses. For instance, together with an analysis that bounds the maximal number of stacks in the evaluation of a program, it gives an overall bound on the memory space needed by the virtual machine. This size-bound can also be used with automatic memory management techniques, e.g. to bound the physical size of regions in region-based systems [18].

We consider bytecode programs that should be executed on a simple stack machine with support for algebraic data types, pattern-matching and tail-recursion. The bytecode language can be the target of the compilation of a simply-typed, first-order functional language. We hint at this functional source language in several places but, since all our results are stated on the bytecode, we do not need to define it formally here (see [1] for a definition). Our *size verification* algorithm is expressed as a static analysis relying on certificates that can be verified at load time. We follow here the usual assumption that code and certificates may be forged by a malicious party. In particular, they do not have to result from the compilation of legit programs. Standard bytecode verification algorithms build for each instruction an abstract representation of the stack. This information typically consists of the types of the values on the stack when the instruction is executed. In a nutshell, the size verification algorithm builds for each instruction an abstract bound on the size of the values in the stack. In our case the bound is a polynomial expression. It also builds proof obligations that the bounds decrease throughout program execution.

Our method generalizes (and lift some of the restrictions) an approach designed for first-order functional languages [1] that relies on a combination of standard techniques for term rewriting systems with a static analysis based on the notion of quasi-interpretation. Similar analyses were also used to deal with systems of concurrent, interactive threads communicating via a shared memory [2]. This paper makes two additional contributions. First, we are able to check programs containing instructions for unconditional jumps and tail-recursive calls, and remove restrictions on the structure of the bytecode that was imposed in these two initial works. These two instructions are essential in the optimization of codes obtained from the compilation of functional programs. They are also required if we need to compile procedural languages. Second, we propose a direct algorithm that depends only on solving a set of arithmetical constraints. Indeed, the size verifications defined in [1,2] are based on a preliminary *shape analysis* which builds, for each bytecode instruction, a sequence of first-order expressions representing the shape of the values in the stack (e.g. it may give the top-most constructors). While the shape verification is well-suited to the analysis of "functional code", it does not scale to programs containing tail recursive calls.

Another result of this work is educational: we present a minimal but still relevant scenario in which problems connected to bytecode verification can be effectively studied. For instance, our virtual machine is based on a set of 8 instructions, a number that has to be compared with the almost 200 opcodes used in the Java Virtual Machine (JVM). We believe that the simplicity of the virtual machine and the bytecode verifiers defined in this paper make them suitable for teaching purposes. (Actually, we have already used them for projects in compiler design classes.)

The paper is organized as follows. Section 2 defines a simple virtual machine and a bytecode language built from a minimal set of instructions. In Section 3, we introduce the notion of quasi-interpretations and define our size verification method. This verification assumes that constructors and function symbols in the bytecode are annotated with suitable functions to bound the size of the values on the stack. Before concluding, we study the complexity of checking the constraints generated during the size analysis. In particular, we show that their satisfiability can be reduced to checking the sign of a polynomial expression.

2 Virtual Machine

We define a simple set of bytecode instructions and a related stack machine. A program is composed of a list of mutually recursive type definitions followed by a list of function definitions. In our setting, a function is a sequence of bytecode instructions. Unlike traditional virtual machines that operate on literal values, such as bytes or floating point numbers, we consider values taken from an arbitrary set of inductive types.

A value v is a term built from a finite set of constructors, ranged over by $\mathsf{c}, \mathsf{d}, \ldots$ The *size* of v, denoted $|v|$, is 0 if v is a constant (a constructor of arity 0) and $1 + \Sigma_{i \in 1..n} |v_i|$ if v is of the form $\mathsf{c}(v_1, \ldots, v_n)$.

We consider a fixed set of type identifiers $t, t', \ldots$ where each identifier is associated to a unique type definition of the form $t = \ldots \mid \mathsf{c}\, \mathit{of}\ t_1 * \cdots * t_n \mid \ldots$ For instance, we can define the type *nat* of natural numbers in unary format and the type *bw* of binary words:

$$\mathit{nat} = \mathsf{z} \mid \mathsf{s}\ \mathit{of}\ \mathit{nat}\ , \qquad\qquad \mathit{bw} = \mathsf{E} \mid \mathsf{O}\ \mathit{of}\ \mathit{bw} \mid \mathsf{I}\ \mathit{of}\ \mathit{bw}\ .$$

For instance, the values $\mathsf{s}(\mathsf{s}(\mathsf{z}))$ of type *nat* and $\mathsf{O}(\mathsf{I}(\mathsf{O}(\mathsf{E})))$ of type *bw* stand for the number 2. We will often use the type *nat* in our examples since functions manipulating natural numbers can be interpreted as an abstraction of functions manipulating finite lists (e.g. addition is related to list catenation).

For the sake of simplicity, we suppose that the code and type of functions is fixed and known in advance. Hence we consider a fixed set of constructor and function names. We suppose that every constructor is declared with its functional type $(t_1, \ldots, t_n) \to t$ and we denote $ar(\mathsf{c})$ the arity of the constructor c. Similar types can be either assigned or inferred for functions. We adopt the notation ε for the empty sequence and $\ell \cdot \ell'$ for the catenation of two sequences. The expression $|\ell|$ denotes the length of ℓ and $\ell[i]$ denotes the i^{th} element in ℓ. When the length is given by the context, we will sometimes use the vectorial notation $\vec{v}$ to represent the sequence $(v_1, \ldots, v_n)$. In the following, we equate a function identifier f with the sequence of instructions of its body code and thus write $f[i]$ for the i^{th} instruction in f.

The virtual machine is built around three components: (1) a *configuration* M that is a stack of call frames; (2) an association list between function identifier and code; (3) a bytecode interpreter, modeled as a reduction relation $M \to M'$. The state of the interpreter, the *configuration* M, is a sequence of frames and we write $M \to M'$ if M reduces to M' using one of the transformation rules described by the table below.

The most important operation of the virtual machine corresponds to function calls. The execution of a function call is represented by a *frame*, that is a triple $(f, pc, \ell)_\rho$ made of a function identifier f, the value of the program counter pc (a natural number in $1..|f|$) and an *evaluation stack* ℓ. A stack is a sequence of values that is used to store both the parameters of the call as well as the "local values" computed during the life span of the frame. Hence the stack partially plays the role devoted to registers in traditional architectures. The annotation ρ is used to keep trace of the call that initiated the frame and has no operational meaning (it is only used to validate our size verification method). We refine the system of annotations in Section 2.1.

We give an informal description of the bytecode language. Let ℓ be the stack of the current frame, i.e. the frame on the top of the current configuration. The instruction `load` i takes a copy of the i^{th} value of ℓ and puts it on the top of the stack (i.e. it is equivalent to a register load). New values may be created using the instruction `build c` n, where `c` is a constructor of arity n. When executed, the n values $v_1, \ldots, v_n$ on top of ℓ are replaced by $\mathsf{c}(v_1, \ldots, v_n)$. The instruction `branch c` j implements a conditional jump on the shape of the value v found on top of ℓ. If v is of the form $\mathsf{c}(v_1, \ldots, v_n)$ then the top of the stack is replaced by the n values $v_1, \ldots, v_n$ (rule BranchThen). Otherwise, the stack is left unchanged and the execution jumps to position j in the code, with $j \in 1..|f|$ (rule BranchElse).

Function calls are implemented by the instruction `call` f n, where n is the arity of f. Upon execution, a fresh call frame is created, which is initialized with a copy of the n values on top of the caller's stack. The lifetime of the current frame is controlled by two instructions: `return` discards the current frame and returns the value on top of the caller's stack; `stop` finishes the execution and returns an error code. Finally, the instruction `jump` j n is an unconditional jump, similar to a goto statement, whereas `tcall` g n is similar to a call instruction, except

Bytecode Interpreter: $M \to M'$

(Load)

$$\frac{f[pc] = \texttt{load}\ i \quad pc < |f| \quad \ell[i] = v}{M \cdot (f, pc, \ell)_\rho \to M \cdot (f, pc+1, \ell \cdot v)_\rho}$$

(Build)

$$\frac{f[pc] = \texttt{build c}\ n \quad pc < |f| \quad ar(\mathsf{c}) = n \qquad \ell = \ell' \cdot (v_1, \dots, v_n) \quad v_o = \mathsf{c}(v_1, \dots, v_n)}{M \cdot (f, pc, \ell)_\rho \to M \cdot (f, pc+1, \ell' \cdot v_o)_\rho}$$

(BranchThen)

$$\frac{f[pc] = \texttt{branch c}\ j \quad pc < |f| \qquad \ell = \ell' \cdot \mathsf{c}(v_1, \dots, v_n)}{M \cdot (f, pc, \ell)_\rho \to M \cdot (f, pc+1, \ell' \cdot (v_1, \dots, v_n))_\rho}$$

(BranchElse)

$$\frac{f[pc] = \texttt{branch c}\ j \quad 1 \leqslant j \leqslant |f| \qquad \ell = \ell' \cdot \mathsf{d}(\dots) \quad \mathsf{c} \neq \mathsf{d}}{M \cdot (f, pc, \ell)_\rho \to M \cdot (f, j, \ell)_\rho}$$

(Call)

$$\frac{f[pc] = \texttt{call}\ g\ n \quad pc < |f| \quad ar(g) = n \qquad \ell = \ell' \cdot \ell'' \quad \ell'' = (v_1, \dots, v_n) \quad \rho' = g(v_1, \dots, v_n)}{M \cdot (f, pc, \ell)_\rho \to M \cdot (f, pc, \ell')_\rho \cdot (g, 1, \ell'')_{\rho'}}$$

(Jump)

$$\frac{f[pc] = \texttt{jump}\ j\ n \quad 1 \leqslant j \leqslant |f| \qquad \ell = \ell' \cdot \ell'' \quad \ell'' = (v_1, \dots, v_n)}{M \cdot (f, pc, \ell)_\rho \to M \cdot (f, j, \ell'')_\rho}$$

(TCall)

$$\frac{f[pc] = \texttt{tcall}\ g\ n \quad pc < |f| \quad ar(g) = n \qquad \ell = \ell' \cdot \ell'' \quad \ell'' = (v_1, \dots, v_n)}{M \cdot (f, pc, \ell)_\rho \to M \cdot (g, 1, \ell'')_\rho}$$

(Stop)

$$\frac{f[pc] = \texttt{stop}}{M \cdot (f, pc, \ell)_\rho \to \mathit{error}}$$

(Return)

$$\frac{f[pc] = \texttt{return} \quad \ell = \ell'' \cdot v_o}{M \cdot (g, pc', \ell')_{\rho'} \cdot (f, pc, \ell)_\rho \to M \cdot (g, pc'+1, \ell' \cdot v_o)_{\rho'}}$$

that the current frame is used to evaluate the call to g. These two instructions are used to share common code between functions and to efficiently compile tail recursion (when call instructions are immediately followed by a return). This is essential because many programming idioms depend heavily on recursion. For example, the Scheme language reference explicitly requires tail recursion to be recognized and automatically optimized by a compiler.

The reduction relation $M \to M'$ is deterministic and uses a special state of the memory, *error*, that denotes the empty configuration ε. The empty state cannot be reached during an execution that does not raise an error (executes a `stop` instruction). A "correct" execution starts with a single frame $M_\iota = (f, 1, \ell)_{f(\ell)}$, where $\ell = (v_1, \dots, v_n)$, and ends with a configuration of the form $M_o = (f, pc, \ell' \cdot v_0)_{f(\ell)}$, where $1 \leqslant pc \leqslant |f|$ and $f[pc] = \texttt{return}$. We name the configuration M_ι a *call to* $f(v_1, \dots, v_n)$ and M_o the *result* of evaluating M_ι and we write $M_o \searrow v_0$. The others cases of blocked configurations are runtime errors.

2.1 Control Flow Graphs and Well-Formedness

Before giving examples of bytecode programs, we define the notions of *control flow graph* (CFG) and *checkpoints* of a function f. The CFG of f is the smallest directed graph $(\{1, \dots, |f|\}, E)$ such that for all node $i \in 1..|f|$ the edge $(i, i+1)$

is in E if $f[i]$ is a `load`, `build`, `call` or `branch` instruction and (i,j) is in E if $f[i]$ is `branch c` j or `jump` j n. Nodes that are the target of a `jump` or `branch` instruction can have several immediate predecessors. We call such nodes the checkpoints of f. The first instruction of a function is also a checkpoint.

We associate to every node i of f the node $PC_i \in 1..|f|$ that is the only checkpoint dominating the node i in the CFG of f: it is the first checkpoint encountered from i when moving backward in the CFG. We say that PC_i is the checkpoint of i and we have $PC_i = i$ iff i is a checkpoint. By construction, every node of a CFG is associated to a unique checkpoint and there is a unique path between PC_i and i without cycles. We also define the predicate $\mathit{Control}_f(i)$ which is true iff i is a checkpoint of f.

We refine the semantics of the virtual machine to take into account checkpoints in frame annotation. We store in the annotations the state of the execution stack when we pass a new checkpoint (together with the state of the stack when the frame is initialized). This improvement is needed for our size analysis but the dynamic semantics of the machine does not need any change. The only difference is in the frame annotation ρ that is now of the form $(g(\ell_o), i, \ell_c)$ where $g(\ell_o)$ is the "call" used to initialize the frame, i is the last checkpoint encountered and ℓ_c is the state of the execution stack when we passed i. For each transition $M \cdot (f, pc, \ell)_\rho \to M' \cdot (f', pc', \ell')_{\rho'}$ of the new relation, we have $\rho' = (\ldots, pc', \ell')$ if pc' is a checkpoint of the function f' and $\rho' = \rho$ otherwise. Note that the evaluation of a `call` or `tcall` instruction (the only case in which $f \neq f'$) always leads to a configuration where the program counter of the last frame is equal to 1, i.e. is a checkpoint.

Annotated Semantics

(Regular)
$$\frac{M \cdot (f, pc, \ell)_{g(\ell_o)} \to M \cdot (f, pc', \ell')_{g(\ell_o)} \qquad \mathit{Control}_f(pc') \text{ is false} \quad \rho = (g(\ell_o), i, \ell_c)}{M \cdot (f, pc, \ell)_\rho \to M \cdot (f, pc', \ell')_\rho}$$

(Checkpoint)
$$\frac{M \cdot (f, pc, \ell)_{g(\ell_o)} \to M' \cdot (f', pc', \ell')_{h(\ell_1)} \qquad \mathit{Control}_f(pc') \text{ is true} \quad \rho = (g(\ell_o), i, \ell_c)}{M \cdot (f, pc, \ell)_\rho \to M' \cdot (f', pc', \ell')_{(h(\ell_1), pc', \ell')}}$$

Examples. Our first example is the function $\mathit{dble} : \mathit{nat} \to \mathit{nat}$, that doubles its parameter. A possible specification of this function using a functional syntax could be $\mathit{dble}(\mathsf{z}) = \mathsf{z}$ and $\mathit{dble}(\mathsf{s}(x)) = \mathsf{s}(\mathsf{s}(\mathit{dble}(x)))$ (actually, the code given below is the result of compiling this functional program as in [1]). In the following, we display the index of each instruction next to its code and underline the indices of checkpoints.

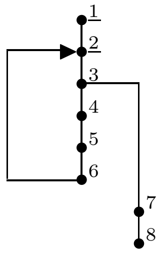
$\mathit{dble} =$ $\underline{1}$	: `load` 1	5	: `branch s` 10	9	: `return`
2	: `branch z` 5	6	: `call` *dble* 1	10	: `stop`
3	: `build z` 0	7	: `build s` 1		
4	: `return`	8	: `build s` 1		

The evaluation of the call to $\mathit{dble}(\mathsf{s}(v))$ gives the following reductions, where w stands for the result of the call to $\mathit{dble}(v)$ (we do not write annotations).

$$(\mathit{dble}, 1, (\mathsf{s}(v))) \to (\mathit{dble}, 2, (\mathsf{s}(v), \mathsf{s}(v))) \to (\mathit{dble}, 5, (\mathsf{s}(v), \mathsf{s}(v))) \to (\mathit{dble}, 6, (\mathsf{s}(v), v))$$
$$\to (\mathit{dble}, 6, (\mathsf{s}(v))) \cdot (\mathit{dble}, 1, (v)) \to \cdots \to (\mathit{dble}, 6, (\mathsf{s}(v))) \cdot (\mathit{dble}, 9, (w))$$
$$\to (\mathit{dble}, 7, (\mathsf{s}(v), w)) \to (\mathit{dble}, 8, (\mathsf{s}(v), \mathsf{s}(w))) \to (\mathit{dble}, 9, (\mathsf{s}(v), \mathsf{s}(\mathsf{s}(w)))) \searrow \mathsf{s}(\mathsf{s}(w))$$

From this simple example we can already see that first-order functional programs admit a direct compilation into our bytecode: every function is compiled into a segment of instructions where pattern matching is represented by a nesting of `branch` instructions. In particular the CFG of a compiled program is a tree. Because our virtual machine does not allow to store code closures, we cannot directly support subroutines or higher-order functions. We plan to study these extensions in future works.

We can simplify our first example following two distinct directions. We obtain an equivalent function by noticing that a natural number that is not of the form `s(...)` is necessarily `z`. Hence we can discard a useless `branch` instruction. A better optimization is obtained with the function *tdble* : *nat* → *nat*: we duplicate the parameter (with a `load` instruction) and use it as an accumulator, giving the opportunity to use a `jump` instruction. Finally, the function *xdble* : *nat* → *nat* is an example of malicious code that loops and computes unbounded values.

inst. # :	*dble*	*xdble*	*tdble*	CFG of *tdble*
1 :	`load` 1	`load` 1	`load` 1	
2 :	`branch s` 6	`build s` 1	`load` 1	
3 :	`call` *dble* 1	`build s` 1	`branch s` 7	
4 :	`build s` 1	`call` *xdble* 1	`load` 2	
5 :	`build s` 1	`return`	`build s` 1	
6 :	`return`		`jump` 2 2	
7 :			`load` 2	
8 :			`return`	

Our last example is the function *sum* : *nat* → *nat* such that a call to $sum(x)$ computes the value of the expression $x + (x - 1) + \cdots + 1$. The function *sum* is interesting because it is a non trivial example mixing recursive calls and "superlinear" size computations: our size verification can be used to prove that the size of the result is bound by $\frac{1}{2}|x|(|x| + 1)$. The definition of *sum* makes use of the function *add* : (*nat*, *nat*) → *nat* that tallies up its two parameters.

sum =	$\underline{1}$: `load` 1	3 : `call` *sum* 1	5 : `return`
	2 : `branch s` 5	4 : `call` *add* 2	

add =	$\underline{1}$: `branch s` 5	4 : `load` 2	7 : `return`
	2 : `load` 1	5 : `jump` 1 2	
	3 : `build s` 1	6 : `load` 1	

Well-Typed Programs. We define a type verification that associates to every bytecode instruction an abstraction of the stack when it is executed. In our case, an abstract state T is a sequence of types $(t_1, \ldots, t_n)$ that matches the types of the values in the stack. We say that a stack ℓ has type T, and we note $\ell : T$, if $\ell = (v_1, \ldots, v_n)$ where v_i is a value of type t_i for all $i \in 1..n$. The type of a function f is a sequence of length $|f|$ of type stacks and a well-typed function

Table 1. Type Analysis ($wt_i(f,\vec{T})$)

Assume $f : (t_1,\ldots,t_n) \to t_0$. Case $f[i]$ of:

(`load` k) then $wt_i(f,\vec{T})$ is true iff $i < |f|$, $T_i[k] = t_k$ and $T_{i+1} = T_i \cdot t_k$;

(`build c` m) let `c` $: (t'_1,\ldots,t'_m) \to t'_0$, then $wt_i(f,\vec{T})$ is true iff $i < |f|$, $T_i = T \cdot (t'_1,\ldots,t'_m)$ and $T_{i+1} = T \cdot t'_0$;

(`branch c` j) let `c` $: (t'_1,\ldots,t'_m) \to t'_0$, then $wt_i(f,\vec{T})$ is true iff $i < |f|$, $j \in 1..|f|$, $T_i = T \cdot t'_0$, $T_{i+1} = T \cdot (t'_1 \ldots t'_m)$ and $T_j = T_i$;

(`call` $g\ m$) let $g : (t'_1,\ldots,t'_m) \to t'_0$, then $wt_i(f,\vec{T})$ is true iff $i < |f|$, $T_i = T \cdot (t'_1 \ldots t'_m)$ and $T_{i+1} = T \cdot t'_0$;

(`tcall` $g\ m$) let $g : (t'_1,\ldots,t'_m) \to t'_0$, then $wt_i(f,\vec{T})$ is true iff $i < |f|$, $t_0 = t'_0$, $T_i = T \cdot (t'_1,\ldots,t'_m)$ and $T_{i+1} = T \cdot t_0$;

(`jump` $j\ m$) then $wt_i(f,\vec{T})$ is true iff $1 \leqslant j \in 1..|f|$, $T_i = T \cdot (t'_1 \ldots t'_m)$ and $T_j = (t'_1 \ldots t'_m)$;

(`return`) then $wt_i(f,\vec{T})$ is true iff $T_i = T \cdot t_0$;

(`stop`) Then $wt_i(f,\vec{T})$ is true.

is a sequence of *well-typed instructions*. The notion of well-typed instruction is formally defined by means of the relation $wt_i(f,\vec{T})$, defined below. For example, if $f[i] =$ `load` k and if the type of $f[i]$ is $T_i = (t_1,\ldots,t_n)$, with $n \geqslant k$, then the type of $f[i+1]$ should be equal to $(t_1,\ldots,t_n,t_k)$. (The abstract state T_i gives the type of values in the stack "before" the execution of instruction i.) A program is well-typed if all its functions are well-typed: a sequence $\vec{T}$ is a *valid abstract execution* of the function $f : (t_1,\ldots,t_n) \to t_0$, denoted $wt(f,\vec{T})$, if and only if $T_1 = (t_1 \ldots t_n)$ and $wt_i(f,\vec{T})$ for all $i \in 1..|f|$. The definition of $wt_i(f,\vec{T})$ is by case analysis on the instruction $f[i]$, see Table 1.

We can define from the predicate wt an algorithm that computes a valid type for a function f if it exists, e.g. using Kildall's algorithm [16]. (We can view type verification as a kind of symbolic execution on stacks of types.) Moreover, we can prove that if the CFG is a connected graph then there is at most one valid type. Then we can assign to every instruction of f the size of its stack, and to every element of that stack a single type. As an example, we give the type inferred for the function $tdble : nat \to nat$.

$\underline{1}$: `load` 1	(nat)	5 : `build s` 1	(nat, nat, nat, nat)
$\underline{2}$: `load` 1	(nat, nat)	6 : `jump` 2 2	(nat, nat, nat, nat)
3 : `branch s` 7	(nat, nat, nat)	7 : `load` 2	(nat, nat, nat)
4 : `load` 2	(nat, nat, nat)	8 : `return`	(nat, nat, nat, nat)

We can prove that the execution of a well-typed program never fails. For example, We can prove a subject reduction property and follow an approach similar to the one used in Section 3 to prove the validity of our size analysis. Due to the limited amount of space available, we prefer to develop the background on size verification, which is the most innovative result of this paper. Most of the formal details on type verification may be found in [1].

The type verification provides a bound on the length of the stacks during an execution and we note $h_{f,i}$ the size of the stack ℓ in a frame $(f, i, \ell)_\rho$ of a well-typed configuration. In the next section we show how to obtain a bound on the size of the computed values from our size analysis. Together, these two information can already be used to reject programs that compute arbitrarily large values but, to obtain a bound on the size needed for the execution of a program, we also need to bound the maximal number of frames, which usually necessitates a termination analysis.

3 Size Verification

We define a size verification based on the notion of quasi-interpretations [12]. This paper makes two additional contributions to our previous works on resource certification [1,2]. First, we are able to check programs whose CFG contains cycles, improving what was done in previous work. Second, we propose a direct algorithm that depends only on solving a set of arithmetical constraints, without resorting to an auxiliary *shape analysis.*

Quasi-interpretation. Quasi-interpretations have been defined by Marion et al. [12] to reason about the implicit complexity of term rewriting systems. The idea is close to polynomial interpretation for termination proofs: we assign to every function and constructor of a program a numerical function bounding the size of the computed values. More formally, a quasi-interpretation assigns to every identifier id in a program a function q_{id} (with arity $ar(id)$) over the non-negative rational numbers $\mathbb{Q}^+$ such that: (1) if c is a constant then $q_{\mathsf{c}}(\,) = 0$; (2) if c is a constructor with arity n then $q_{\mathsf{c}}(x_1, \ldots, x_n) = d + \Sigma_{i\in 1..n} x_i$, where $d \geqslant 1$; (3) if f is a function with arity n then $q_f : (\mathbb{Q}^+)^n \to \mathbb{Q}^+$ is monotonic and for all $i \in 1..n$ we have $q_f(x_1, \ldots, x_n) \geqslant x_i$.

An assignment can be easily extended to functional expressions as follows: $q_x = x$; $q_{\mathsf{c}(e_1,\ldots,e_n)} = q_{\mathsf{c}}(q_{e_1}, \ldots, q_{e_n})$; and $q_{f(e_1,\ldots,e_n)} = q_f(q_{e_1}, \ldots, q_{e_n})$. Then an assignment is a valid quasi-interpretation for a system of recursive function definitions if for all declarations $f(p_1, \ldots, p_n) = e$ in the program, the inequality $q_{f(p_1,\ldots,p_n)} \geqslant q_e$ holds. For instance, if we choose $q_{\mathsf{s}} = 1 + x$ for the quasi-interpretation of the constructor in *nat* (by definition, $q_{\mathsf{z}} = 0$) then $q_{dble}(x) = 2x$ is a valid quasi-interpretation for the function *dble* defined in our examples: we have $q_{dble}(q_{\mathsf{z}}(\,)) \geqslant q_{\mathsf{z}}(\,)$ and $q_{dble}(q_{\mathsf{s}}(x)) \geqslant q_{\mathsf{s}}(q_{\mathsf{s}}(q_{dble}(x)))$. In general, a quasi-interpretation provides a bound on the size of the computed values as a function of the size of the input data. If $f(v_1, \ldots, v_n) \searrow v$ then $|v| \leqslant q_v \leqslant q_f(|v_1|, \ldots, |v_n|)$.

The problem of synthesizing quasi-interpretations (from a set of functional declarations) is connected to the synthesis of polynomial interpretations for termination but it is generally easier because inequalities do not need to be strict and small degree polynomials are often enough. For instance, Amadio [3,4] has considered the problem of automatically inferring quasi-interpretations in the space of multi-variate max-plus polynomials.

In this paper, we define a similar notion of quasi-interpretation for bytecode programs. Assume a function f of the bytecode program. An assignment associates to every checkpoint i of f a polynomial expression $q_{f,i}$ with $h_{f,i}$ variables. We also use the notation q_f to denote the function $q_{f,1}$ assigned to the entry point of f. Like in the functional case, we require that each polynomial $q_{f,i}$ satisfies the hypotheses for quasi-interpretations (properties (1)-(3) listed above). The machine-checkable certificates used in our size verification are quasi-interpretations, that is assignment of numerical functions, in our case polynomial expressions, to instructions in the program. An advantage of this approach is that quasi-interpretation can be synthesized at the source-code level and verified at the bytecode level. We will not address synthesis issues in this paper and we suppose that the bytecode comes with all the necessary types and size annotations. For example the function *tdble* has two checkpoints, the nodes 1 and 2, with respective types (nat) and (nat, nat), which means that $h_{tdble,1} = 1$ and $h_{tdble,2} = 2$. In the following we assume that the assignment is $q_{tdble,1}(x_1) = 2x_1$ and $q_{tdble,2}(y_1, y_2) = y_1 + y_2$.

Size Analysis. We show how to check the validity of an assignment and to obtain a size bound from a quasi-interpretation. Like the type verification, our *size verification* associates to every bytecode instruction an abstraction of the stack at the time it is executed. In this case, the abstraction is a combination of a sequence of *size variables*, which stands for the best size bounds we can obtain, together with arithmetic constraints between these variables. Contrary to the size verification defined in [1], we directly infer a size bound, without using an auxiliary "shape verification" (that is a static analysis which provides partial informations on the structure of the elements in the stack). The advantage of a direct approach is to get rid of the restrictions imposed by the shape analysis, especially: (1) that the CFG of functions must be a tree and (2) that along each execution path, we must not have a `branch` instruction after a `call` instruction.

We suppose that the bytecode is well-typed, which means that we know the number $h_{f,i}$ of elements on the stack before executing the instruction $f[i]$. We associate to each checkpoint i of the function f: (1) a sequence of fresh (size) variables $\vec{x}_{f,i} =_{\text{def}} (x_1, \dots, x_{h_{f,i}})$ and (2) a polynomial expression $q_{f,i}$ with variables $\vec{x}_{f,i}$ and coefficients in $\mathbb{Q}$.

The size analysis is formally defined by the predicate $wsz_i(f, \vec{S}, \vec{\Phi})$ given in Table 2. The definition of $wsz_i(f, \vec{S}, \vec{\Phi})$ is by case analysis on the instruction $f[i]$ and expresses that (1) the size of every element on the stack at instruction i is bounded by the expression $q_{f,PC_i}(\vec{x}_{f,PC_i})$, and (2) the quasi-interpretations decrease every time we pass a new checkpoint. We say that the size analysis is successful if there are two sequences $\vec{S} = (S_1, \dots, S_{|f|})$ and $\vec{\Phi} = (\Phi_1, \dots, \Phi_{|f|})$ such that $wsz_i(f, \vec{S}, \vec{\Phi})$ for all $i \in 1..|f|$, and $S_i = \vec{x}_{f,i}$ and $\Phi_i = \emptyset$ if i is a checkpoint of f. We note this relation $wsz(f, \vec{S}, \vec{\Phi})$.

Table 2. Size Analysis ($wsz_i(f, \vec{S}, \vec{\Phi})$)

Let $j = PC_i$ be the checkpoint of i. Case $f[i]$ of:

(`load` k) let x_k be the k^{th} variable of S_i, and x a fresh size variable. If $Control(i{+}1)$ then $wsz_i(f, \vec{S}, \vec{\Phi})$ iff the formula $\psi_{succ} =_{\text{def}} \big(\Phi_i \land x = x_k\big) \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{f,i+1}(S_i \cdot x)\big)$ is a tautology. Otherwise $wsz_i(f, \vec{S}, \vec{\Phi})$ iff $\big(S_{i+1} = S_i \cdot x\big)$ and $\big(\Phi_{i+1} = \Phi_i \land x = x_k\big)$.

(`build c` n) Assume $n = ar(\mathsf{c})$ and $S_i = S' \cdot (x_1, \ldots, x_n)$, and let x_0 be a fresh size variable. First we check the validity of $\psi_{build} =_{\text{def}} \Phi_i \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{\mathsf{c}}(x_1, \ldots, x_n)\big)$. If $Control(i + 1)$ then $wsz_i(f, \vec{S}, \vec{\Phi})$ iff the formula $\psi_{succ} =_{\text{def}} \big(\Phi_i \land x_0 = q_{\mathsf{c}}(x_1, \ldots, x_n)\big) \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{f,i+1}(S' \cdot x_0)\big)$ is a tautology. Otherwise $wsz_i(f, \vec{S}, \vec{\Phi})$ iff $\big(S_{i+1} = S' \cdot x_0\big)$ and $\big(\Phi_{i+1} = \Phi_i \land x_0 = q_{\mathsf{c}}(x_1, \ldots, x_n)\big)$.

(`branch c` k) Assume $n = ar(\mathsf{c})$ and $S_i = S' \cdot x_0$, and let $x_1, \ldots, x_n$ be fresh size variables. The predicate $wsz_i(f, \vec{S}, \vec{\Phi})$ is true iff the following two conditions are true (one condition for each successor of i in f).

(C1) if $Control(i + 1)$ then $\psi_{then} =_{\text{def}} \big(\Phi_i \land x_0 = q_{\mathsf{c}}(x_1, \ldots, x_n)\big) \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{f,i+1}(S' \cdot (x_1 \ldots x_n))\big)$ is a tautology otherwise $\big(S_{i+1} = S' \cdot (x_1 \ldots x_n)\big)$ and $\big(\Phi_{i+1} = \Phi_i \land x_0 = q_{\mathsf{c}}(x_1, \ldots, x_n)\big)$.
(C2) if $Control(k)$ then $\psi_{else} =_{\text{def}} \Phi_i \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{f,k}(S_i)\big)$ is a tautology otherwise $\big(S_k = S_i\big)$ and $\big(\Phi_k = \Phi_i\big)$.

(`call` g n) Assume $n = ar(g)$ and $S_i = S' \cdot (x_1, \ldots, x_n)$ and let x_0 be a fresh size variable. First we check the validity of $\psi_{call} =_{\text{def}} \Phi_i \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{g,1}(x_1, \ldots, x_n)\big)$. If $Control(i + 1)$ then $wsz_i(f, \vec{S}, \vec{\Phi})$ iff the formula $\psi_{succ} =_{\text{def}} \big(\Phi_i \land x_0 \leqslant q_{g,1}(x_1, \ldots, x_n)\big) \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{f,i+1}(S' \cdot x_0)\big)$ is a tautology. Otherwise $wsz_i(f, \vec{S}, \vec{\Phi})$ iff $\big(S_{i+1} = S' \cdot x_0\big)$ and $\big(\Phi_{i+1} = \Phi_i \land x_0 \leqslant q_{g,1}(x_1, \ldots, x_n)\big)$.

(`tcall` g n) Assume $n = ar(g)$ and $S_i = S' \cdot (x_1, \ldots, x_n)$ and let x_0 be a fresh size variable. The predicate $wsz_i(f, \vec{S}, \vec{\Phi})$ is true iff the formula ψ_{tcall} is valid, where $\psi_{tcall} =_{\text{def}} \Phi_i \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{g,1}(x_1, \ldots, x_n)\big)$.

(`jump` k n) Assume $S_i = S' \cdot (x_1, \ldots, x_n)$. If $Control(k)$ then $wsz_i(f, \vec{S}, \vec{\Phi})$ iff the formula $\psi_{succ} =_{\text{def}} \Phi_i \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{f,k}(x_1 \ldots x_n)\big)$ is a tautology. Otherwise $wsz_i(f, \vec{S}, \vec{\Phi})$ iff $\big(S_k = (x_1, \ldots, x_n)\big)$ and $\big(\Phi_k = \Phi_i\big)$.

(`stop` or `return`) Then the predicate $wsz_i(f, \vec{S}, \vec{\Phi})$ is true.

The size analysis is compositional (we only need to analyze each functions separately) and always terminates (since every instruction is visited at most once). The size analysis for a function f associates to every instruction i of f a sequence of variables of size $h_{f,i}$, denoted S_i, and a set of constraints between linear combinations of these variables, denoted Φ_i. Intuitively, the k^{th} variable of S_i is a bound on the size of the k^{th} element of the execution stack when the instruction $f[i]$ is executed, while Φ_i contains valid constraints between the bounds. For example, if $f[i] =$ `load` k and $S_i = (x_1, \ldots, x_n)$ we impose that $S_{i+1} = S_i \cdot x$ and that Φ_{i+1} implies $x = x_k$, meaning that we add a value on top of the stack ℓ, whose size is bounded by x_k, our best known bound on the size of the k^{th} value in ℓ. The analysis generates also a set of *proof obligations*, $\psi_{succ}, \psi_{call}, \ldots$ which are arithmetic formulas that should be checked in order to prove the validity of the size certificates (i.e. the quasi-interpretation). We say that the formula $\Phi \Rightarrow p(\vec{x}) \geqslant q(\vec{y})$ with free size variables $\vec{y}$ is a tautology if for all valuation σ from $\vec{y}$ to positive natural numbers such that $\sigma(\Phi)$ is true then the inequality $\sigma(p(\vec{x}) \geqslant \sigma(q(\vec{y}))$ is true.

We show the result of the size analysis for our running example, *tdble*. We give in front of each instruction the size stack S_i and the constraint Φ_i such that $wsz(f, \vec{S}, \vec{\Phi})$. Then we check the validity of the various auxiliary conditions: there is one condition to prove each time the successor of an instruction is a checkpoint and one condition to prove for each `build`, `call` and `tcall` instruction.

$\underline{1}$: `load` 1	x_1	$\emptyset$
$\underline{2}$: `load` 1	$y_1\ y_2$	$\emptyset$
3 : `branch s` 7	$y_1\ y_2\ z_1$	$(z_1 = y_1)$
4 : `load` 2	$y_1\ y_2\ z_2$	$(z_1 = y_1) \wedge (z_1 = z_2 + 1)$
5 : `build s` 1	$y_1\ y_2\ z_2\ z_3$	$(z_1 = y_1) \wedge (z_1 = z_2 + 1) \wedge (z_3 = y_2)$
6 : `jump` 2 2	$y_1\ y_2\ z_2\ z_4$	$(z_1 = y_1) \wedge (z_1 = z_2 + 1) \wedge (z_3 = y_2) \wedge (z_4 = z_3 + 1)$
7 : `load` 2	$y_1\ y_2\ z_1$	$(z_1 = y_1)$
8 : `return`	$y_1\ y_2\ z_1\ z_5$	$(z_1 = y_1) \wedge (z_5 = y_2)$

We need to check three proof obligations in the size verification of *tdble*. The first formula corresponds to the `build` instruction $\psi_5 = \Phi_5 \Rightarrow \big(q_{tdble,2}(y_1, y_2) \geqslant q_{\mathsf{s}}(z_3)\big)$. The two others formulas correspond to the possible transitions to checkpoint 2 (from instructions 1 and 6) which gives $\psi_1 =_{\text{def}} \Phi_1 \Rightarrow \big(q_{tdble,1}(x_1) \geqslant q_{tdble,2}(x_1, x_1)\big)$ and $\psi_6 =_{\text{def}} \Phi_6 \Rightarrow \big(q_{tdble,2}(y_1, y_2) \geqslant q_{tdble,2}(z_2, z_4)\big)$. Once simplified, we can easily show that these constraints are valid: ψ_5 is equivalent to $(z_1 = y_1) \wedge (z_1 = z_2 + 1) \wedge (z_3 = y_2) \Rightarrow y_1 + y_2 \geqslant z_3 + 1$, while $\psi_1 \equiv 2x_1 \geqslant x_1 + x_1$ and $\psi_6 \equiv (y_1 = z_2 + 1) \wedge (z_4 = y_2 + 1) \Rightarrow (y_1 + y_2 \geqslant z_2 + z_4)$.

Next, we show the result of the size analysis for the function *sum*. We assume that the quasi-interpretations of *sum* and *add* are the functions $q_{sum}(x) = \frac{1}{2}x(x+1)$ and $q_{add}(x, y) = x + y$.Instruction 5 of *sum* is a checkpoint and we assume that $q_{sum,5}(x) = x$.

$\underline{1}$: `load` 1	x_1	$\emptyset$
2 : `branch s` 5	$x_1\ x_2$	$(x_2 = x_1)$
3 : `call` *sum* 1	$x_1\ y_1$	$(x_2 = x_1) \wedge (x_2 = y_1 + 1)$
4 : `call` *add* 2	$x_1\ y_2$	$(x_2 = x_1) \wedge (x_2 = y_1 + 1) \wedge (y_2 \leqslant q_{sum}(y_1))$
$\underline{5}$: `return`	z_1	$\emptyset$

The analysis of *sum* (note that the functions *add* and *sum* may be analysed separately) gives only two non-trivial proof obligations that are related to the two `call` instructions in the code. These formulas are $\psi_3 = \Phi_3 \Rightarrow \big(q_{sum}(x_1) \geqslant q_{sum}(y_1)\big)$ and $\psi_4 = \Phi_4 \Rightarrow \big(q_{sum}(x_1) \geqslant q_{add}(x_1, y_2)\big)$. Once simplified, we can easily show that these constraints are valid: ψ_3 is equivalent to $q_{sum}(y_1 + 1) \geqslant q_{sum}(y_1)$, while ψ_4 is a consequence of $q_{sum}(y_1 + 1) \geqslant q_{sum}(y_1) + y_1 + 1$.

Finally, we can check that the function *xdble*, our example of malicious code, does not succeed the size analysis. Let us consider the proof obligations generated in the analysis of the function *xdble*:

$\underline{1}$: `load` 1	x_1	$\emptyset$
2 : `build s` 1	$x_1\ x_2$	$x_2 = x_1$
3 : `build s` 1	$x_1\ x_3$	$x_2 = x_1 \wedge x_3 = x_2 + 1$
4 : `call` *xdble* 1	$x_1\ x_4$	$x_2 = x_1 \wedge \cdots \wedge x_4 = x_3 + 1$
5 : `return`	$x_1\ x_5$	$x_2 = x_1 \wedge \cdots \wedge x_5 \leqslant q_{xdble,1}(x_4)$

The condition corresponding to instruction 4, the only `call` instruction, is $\Phi[4] \Rightarrow q_{xdble,1}(x_1) \geqslant q_{xdble,1}(x_4)$, that is equivalent to $x_4 = x_1 + 2 \Rightarrow q_{xdble,1}(x_1) \geqslant q_{xdble,1}(x_4)$, which is obviously not satisfiable since $q_{xdble,1}(x)$ is monotone.

Deriving Size Bounds from the Size Analysis. We prove that if the size analysis returns a solution for all the functions of a program, then we can extract a bound on the size of the values computed during the execution. In order to prove this property, we extend the predicate *wsz* to frames and then configurations of the virtual machine.

Assume $wsz(f, \vec{S}, \vec{\Phi})$ and let ρ be the annotation $(g(\ell_o), k, (v_1 \dots v_n))$. We say that the frame $(f, i, \ell)_\rho$ is *well-sized* if $q_{f,k}(q_{v_1}, \dots, q_{v_n})$ bounds the size of all the values in ℓ and if the constraint Φ_i is verified when we replace the variables of S_i by the quasi interpretation of the corresponding values in ℓ and the variables of $\vec{x}_{f,k}$ by the values $q_{v_1}, \dots, q_{v_n}$. We denote this last property $(f, i, \ell)_\rho \models (\vec{S}, \vec{\Phi})$. Then we say that the configuration M is well-sized if all the frames in M are well-sized and if the quasi-interpretations of the checkpoints decrease, see the table below which defines the relation $wsz(M)$. We introduce some auxiliary notations to help us define the relation *wsz* formally. Assume $wsz(f, \vec{S}, \vec{\Phi})$ and let $(f, i, \ell)_\rho$ be a frame such that ρ is the annotation $(g(\ell_o), k, \ell_c)$. Assume $\ell_o = (u_1, \dots, u_n)$ and $\ell_c = (u'_1, \dots, u'_m)$. We define the two expressions $\hat{q}(\rho)$ and $q(f, \rho)$ as follows: $\hat{q}(\rho) =_{\text{def}} q_{g,1}(q_{u_1}, \dots, q_{u_n})$ and $q(f, \rho) =_{\text{def}} q_{f,k}(q_{u'_1}, \dots, q_{u'_m})$. The value of $\hat{q}(\rho)$ denotes the best size bound known when the frame is initialized, while $q(f, \rho)$ denotes the best size bound known when we reached the last checkpoint. Let $\ell = (v_1, \dots, v_n)$ be a sequence of values and $\vec{x} = (x_1, \dots, x_n)$ a sequence of variables of the same length. We write $\left[{}^{\ell}/_{\vec{x}}\right]^{||}$ the substitution $\left[{}^{q_{v_1}}/_{x_1}\right] \dots \left[{}^{q_{v_n}}/_{x_n}\right]$. The constraint $\vec{\Phi}$ is true for the frame $(f, i, \ell)_\rho$, denoted $(f, i, \ell)_\rho \models (\vec{S}, \vec{\Phi})$ if and only if the constraint $\Phi_i\left[{}^{\ell}/_{S_i}\right]^{||}\left[{}^{\ell_c}/_{\vec{x}_{f,k}}\right]^{||}$ is valid.

Well-Sized Configurations: $wsz(M)$

$$\frac{\begin{array}{c} wsz(f, \vec{S}, \vec{\Phi}) \\ (f, pc, \ell)_\rho \models (\vec{S}, \vec{\Phi}) \quad \ell = (v_1, \dots, v_n) \\ \hat{q}(\rho) \geqslant q(f, \rho) \geqslant |v_i| \qquad i \in 1..n \end{array}}{wsz(f, pc, \ell)_\rho}
\qquad
\frac{\begin{array}{c} M \equiv (f_1, i_1, \ell_1)_{\rho_1} \dots (f_m, i_m, \ell_m)_{\rho_m} \\ \ell^o_k = arg(M, k+1) \quad wsz(f_k, i_k, \ell_k \cdot \ell^o_k)_{\rho_k} \\ wsz(f_m, i_m, \ell_m)_{\rho_m} \quad q(f_j, \rho_j) \geqslant q(f_{j+1}, \rho_{j+1}) \\ k \in 1..m-1 \quad j \in 1..m-1 \end{array}}{wsz(M)}$$

We can show that the predicate *wsz* is preserved by reduction.

Theorem 1 (Preservation). *If $wsz(M)$ and $M \to M'$ then $wsz(M')$.*

Proof. By induction on the derivation of $M \to M'$, see [8] for a detailed proof.

A corollary of this result is that for every program succeeding the size analysis, if the initial configuration $(f, 1, (v_1 \dots v_n))$ is well-sized then the values computed during the execution are bounded by $q_f(q_{v_1}, \dots, q_{v_n})$.

Theorem 2 (Size Bound). *Assume f is a function in a program that succeeds the size analysis. If the initial configuration $(f, 1, (v_1 \dots v_n))$ reduces to M then for all value v occurring in a frame of M we have $|v| \leqslant q_f(q_{v_1}, \dots, q_{v_n})$.*

Proof. Let ℓ be a stack of the form $(v_1, \dots, v_n)$. By hypothesis we have $(f, 1, \ell)_\rho \to^* M$ with $\rho = (f(\ell), 1, \ell)$ and $M \equiv (f_1, i_1, \ell_1)_{\rho_1} \ \dots \ (f_m, i_m, \ell_m)_{\rho_m}$, where ρ_1 is of the form $(f(\ell), PC_{i_1}, \ell_c^1)$. By Theorem 1 we have that M is well-sized, that is $wsz(M)$. Hence (1) $q(f_j, \rho_j) \geqslant q(f_{j+1}, \rho_{j+1})$ for all $j \in 1..m-1$ and (2) $wsz(f_k, i_k, \ell_k \cdot \ell_k^o)_{\rho_k}$ for all $k \in 1..m-1$ and $wsz(f_m, i_m, \ell_m)_{\rho_m}$. By property (2) and definition of the predicate *wsz* on frames, $|v| \leqslant q(f_k, \rho_k) \leqslant \hat{q}(\rho_k)$ for all value v occurring in the k^{th} frame of M and by property (1) we obtain that $|v| \leqslant q(f_1, \rho_1) \leqslant \hat{q}(\rho_1) = q_f(q_{v_1}, \dots, q_{v_n})$, as needed.

4 Solving Size Constraints

Size verification generates a system of auxiliary arithmetical constraints that we need to solve. On the whole, each constraint is of the form $\Phi \Rightarrow p(\vec{x}) \leqslant q(\vec{y})$, where Φ is a conjunction of equality and inequality constraints (see the discussion below) and p, q are polynomial expressions with coefficients in $\mathbb{Q}$. A constraint is generated for each `build`, `call` and `tcall` instruction and for each transition from an instruction to a checkpoint. In this section we study the problem of checking the validity of these constraints and show that we can always reduce to the problem of checking the sign of a polynomial expression.

We start by partitioning the set V of all size variables used in the size verification. We define the sets $V_{\texttt{load}}$, $V_{\texttt{build}}$, $V_{\texttt{branch}}$ and $V_{\texttt{call}}$ of variables that were introduced respectively when checking a `load`, `build`, `branch` and a `call` or a `tcall` instruction. We also define the set V_o of all variables associated to checkpoints. To simplify our result, we assume that `branch` instructions never act on variables in $V_{\texttt{build}}$ (and transitively on variables introduced by a `load` instruction that corresponds to a variable of $V_{\texttt{build}}$). Intuitively, this corresponds to forbid cases where a `branch` instruction is applied to a value whose head constructor is known at compile time (indeed it is possible to trace back the `build` instruction that created it). A consequence of this assumption is to avoid "dead-code", i.e. a part of the code that cannot be reached during an execution.

A brief inspection of the definition of *wsz* shows that the proof obligations generated during the size analysis are all of the form $\Phi \Rightarrow q_f(\vec{x}_f) \geqslant$

$q_g(y_1,\ldots,y_n)$, where $\vec{x}_f$ is a vector of fresh variables and Φ is a conjunction of atoms of the form:

(Load)	$y = x$	where $y \in V_{\texttt{load}}$
(Build)	$y = \sum_{i\in I} x_i + d$	where $y \in V_{\texttt{build}}$, and $d \geqslant 1$
(Branch)	$\sum_{i\in I} y_i + d = x$	where $y_i \in V_{\texttt{branch}}$ for all $i \in I$ and $d \geqslant 1$
(Call)	$y \leqslant q(x_1,\ldots,x_n)$	where $y \in V_{\texttt{call}}$ and q is a polynomial expression with the properties of quasi-interpretations.

We can solve this kind of constraints using the following simple steps:

– first, we eliminate the variables of $V_{\texttt{load}}$ and $V_{\texttt{build}}$ by substitution. This step eliminates the constraints of type (Load) and (Build). The system resulting after this step is made up of (Branch) and (Call) constraints and all the remaining variables are in $V \setminus (V_{\texttt{load}} \cup V_{\texttt{build}})$;

– then we use the hypothesis that we never apply a `branch` instruction on a value introduced by a `build`. So we can replace every (Branch) constraint by a simple substitution. Hence all the constraints of the resulting system are of type (Call) with variables in $V_o \cup V_{\texttt{call}}$;

– finally we are left to check an inequality of the form $\sigma(g(y_1,\ldots,y_n)) \leqslant \sigma(f(\vec{x}_f))$ where σ is the substitution obtained after the first two steps. By construction there are no variables of $V_{\texttt{call}}$ in $\sigma(f(\vec{x}_f))$. Let $C_1,\ldots,C_m$ be the remaining (Call) constraints. For every $i \in 1..m$ the constraint C_i is of the kind $z_i \leqslant p(\vec{a_i})$. Since there are no variables of $V_{\texttt{call}}$ in $\sigma(f(\vec{x}_f))$ we can simply check the inequality after replacing the occurrences of z_i by the expression $p(\vec{a_i})$ (since we work with quasi-interpretation the function p is monotone). Hence it is equivalent to check the sign of the (polynomial) expression: $\big(f(\vec{x}_f) - g(y_1,\ldots,y_n)\big)(\sigma \circ \left[{}^{p(\vec{a_i})}/_{z_i}\right]_{i\in 1..m})$.

5 Conclusion and Related Work

Ensuring bounds on the resources needed for executing a program is a critical safety property. In this paper, we define a new "size analysis" and show how to derive a bound on the size of the values computed by a program. This method has several advantages. The size-bound obtained with our approach is a polynomial expression on the size of the input parameters of the program. Also, programs can be analyzed incrementally (each function is analyzed separately), at the level of the bytecode. These features are particularly interesting in the context of mobile code applications, in which programs can be dynamically loaded from untrusted, possibly malicious sites.

The problem of bounding computational resources has already attracted considerable attention. Many works have focused on (first-order) functional languages starting from Cobham's characterization of polynomial time functions by bounded recursion on notation [6]. Following works, see e.g. [5,9,10], have developed various inference techniques that allow for efficient analyses while capturing a sufficiently large range of practical algorithms. None of these works have been applied to bytecode languages. Actually, most of the researches on bytecode verification tends to concentrate on the integrity of the execution environment. We have presented in [1] a virtual machine and a corresponding bytecode for a first-order functional language and shown how size and termination annotations can be formulated and verified at the level of the bytecode. In this paper, we extend this language with instructions for "tail recursive" calls and unconditional jumps, which are vital to implement common program optimizations. In particular, we can analyze bytecode sequences whose control flow graph includes cycles, whereas the size analysis defined in [1] can only handle tree shaped control flow graphs. Work on resource bounds for "Java-like" bytecode languages is carried out in the MRG project [15]. One main technical difference is that they rely on a general proof carrying code approach while we follow a Typed Assembly Language (TAL) approach. Also, their analysis focuses on the size of the heap while we only consider stack allocated values. Crary and Weirich [7] define a TAL for resource bound certification. Their approach is based on a dependent type-system where types include a "resource skeleton", that is a set of functions (expressed in a ML-like language) computing the resource behavior of the program. Resource skeleton cannot be inferred and should be written by the programmer. Another related work is due to Marion and Moyen [13] who define a resource analysis for counter machines by reduction to a certain type of termination in Petri Nets. Their virtual machine is much more restricted than the one studied here: natural numbers is the only data type and the stack can only contain return addresses.

We are currently experimenting with the automatic derivation of quasi-interpretation at the bytecode level. At the moment, we only have methods to infer quasi-interpretations (with max-plus polynomials) from functional code [4]. Plans for future works also include extending our approach to a more complicated virtual machine, e.g. with support for objects (as in the Java Virtual Machine), heap references or subroutines.

References

1. R. Amadio, S. Coupet-Grimal, S. Dal Zilio and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. In *International Conference on Computer Science Logic (CSL)*, LNCS, vol. 3210, Springer, 2004.
2. R. Amadio and S. Dal Zilio. Resource control for synchronous cooperative threads. In *15th International Conference on Concurrency Theory (CONCUR)*, LNCS vol. 3170, Springer, 2004.
3. R. Amadio. Max-plus quasi-interpretations. In *6th International Conference on Typed Lambda Calculi and Applications (TLCA)*, LNCS vol. 2701, Springer, 2003.
4. R. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1-2):29–60, 2005.
5. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
6. A. Cobham. The intrinsic computational difficulty of functions. In *Proceedings Logic, Methodology, and Philosophy of Science II*, North Holland, 1965.
7. K. Crary and S. Weirich. Resource bound certification. In *Principles of Programming Languages (POPL).* ACM, 2000.

8. S. Dal Zilio and R. Gascon. Resource Bound Certification for a Tail-Recusive Virtual Machine. LIF Research Report 26, 2005.
9. M. Hofmann. The strength of non size-increasing computation. In *Principles of Programming Languages (POPL)*, ACM, 2002.
10. N. Jones. *Computability and complexity, from a programming perspective.* MIT-Press, 1997.
11. T. Lindholm and F. Yellin. *The Java virtual machine specification.* Addison-Wesley, 1999.
12. J.-Y. Marion. *Complexité implicite des calculs, de la théorie à la pratique.* Habilitation à diriger des recherches, Université de Nancy, 2000.
13. J.-Y. Marion and J.-Y. Moyen. *Termination and resource analysis of assembly programs by Petri Nets.* Technical Report, Université de Nancy, 2003.
14. G. Morriset, D. Walker, K. Crary and N. Glew. From system F to typed assembly language. In *ACM Transactions on Programming Languages and Systems*, 21(3):528-569, 1999.
15. D. Sannella. Mobile Resource Guarantee. IST-Global Computing research project, 2001. `http://www.dcs.ed.ac.uk/home/mrg/`.
16. G. Kildall. A unified approach to global program optimization. In *Principles of Programming Languages (POPL).* ACM, 1973.
17. G Necula. Proof-carrying code. In *Principles of Programming Languages (POPL).* ACM, 1997.
18. M. Tofte and J.-P. Talpin. Region-Based Memory Management. In *Information and Computation*, 132(2):109–176, 1997.

A Path Sensitive Type System for Resource Usage Verification of C Like Languages

Hyun-Goo Kang, Youil Kim, Taisook Han, and Hwansoo Han

Department of Computer Science,
Korea Advanced Institute of Science and Technology
{hgkang,youil.kim}@ropas.kaist.ac.kr, {han,hshan}@cs.kaist.ac.kr

Abstract. In this paper, we present a path sensitive type system for resource usage verification. Path sensitivity is essential to model resource usage in C programs correctly and accurately. So far, most of methods to analyze this kind of property in the path sensitive way have been proposed as whole program analyses or unsound analyses. Our main contributions are as follows. First, we formalize a sound analysis for path sensitive resource usage properties in C like languages. To the best of our knowledge, it is the first sound and modular analysis for this problem. We provide the complete proof for the soundness of the type system and algorithm. Second, our analysis is modular, and we provide an inference algorithm to generate function summaries automatically. We believe that our approach suggests new insights into the design of modular analyses.

1 Introduction

It is an important program correctness criterion that a program uses resources in valid manner. A resource is an object that must be used according to a well-defined protocol; such protocol is usually specified by correct sequences of actions on the resource, recognizable by a finite state machine (FSM). For example, a file should be open before being written. A memory cell should not be accessed after deallocation. A lock acquired should be released eventually.

In verification of resource usage protocols, the program analysis should be path sensitive in order to trace the resource states accurately. For example, consider the following C code fragments:

```
[ Program 1 ]                  [ Program 2 ]
main() {                       main() {
  FILE* fp = fopen("f","w");     FILE* fp = fopen("f","w");
  fprintf(fp,"x");               if (fp)
  fclose(fp);                    { fprintf(fp,"x"); fclose(fp); }
}                              }
```

In the path where `fopen` returns a non-zero value, the file is open, but if the return value is 0, the file is still closed. Thus, Program 1 is unsafe on the path where `fopen` returns 0 as `fprintf` may access the closed file. On the other hand, Program 2 is correct since those two paths are distinguished by the branch condition. If a program analyzer ignores the return value of `fopen` and incorrectly assumes that `fopen` always opens the specified file, we may get undesirable results for both programs; the analyzer would miss the bug in Program 1, and it would produce a false alarm for Program 2 because it concludes that the file remains open along with the false branch.

So far, a number of static analyses on resource usage protocols have been proposed in various contexts. In the approaches of [1,17,10,6,14,21,28], they do not present any concrete method to identify actions which change the state of a resource. In other words, actions are limited to syntactically identifiable set. It is not clear whether they can be easily extended to identify actions dependent on values of variables as shown in the example programs. In the approaches of [5,2,16,13], they can identify actions in cooperation with values of variables (path sensitive). All of these works are proposed as whole program analyses. In contrast to sound approaches mentioned above, there are several approaches which are aimed to detect bugs effectively at the cost of unsoundness, which include resource usage protocols [8,3,31]. In summary, there is no sound program analysis which can verify the resource usage property in path sensitive and modular way.

Based on the observations above, our aims are

– To design a resource usage analysis for C like languages that is effective for the path sensitive resource usage.
– To design it as a modular analysis that enables the modular specification and verification of resource usage. Also, we expect to overcome the scalability problem of whole program analysis with modularity.
– To design it as a sound analysis for the purpose of proving the *absence* of resource usage bugs.

Main contributions of this paper are as follows:

– We formalize a sound analysis for path sensitive resource usage properties in C like languages.
– We formalize the analysis as a type system, which gives us insights for designing modular analyses; we devise a function summary mechanism that is expressive enough in the presence of flow/context/path sensitivity and automatically inferable.

The remainder of this paper is organized as follows: In Section 2, we give an overview of our approach. The syntax and dynamic semantics of the language are described in Section 3. In Section 4, we present the type system to analyze resource usage in modular and path sensitive way. In Section 5, the type inference algorithm is presented. We relate our work with previous research in Section 6 and conclude in Section 7.

2 Overview of Our Approach

We have examined intensively the file resource usage appearing in SPEC benchmark suite[26], which include gcc packages used as experimental data in a related work ESP[5].

Here are some observations we learned from the code:

- The path sensitivity mentioned in Section 1 is a common idiom in designing and using API (Application Programmer Interface) in C. So, the path sensitivity is essential to model resource usage in C correctly and accurately.
- A pointer to file-like resources is normally used just as a reference. For example, consider file pointer `fp` in program 2. Programmer does not assign any new value in `*fp` directly, but just use it (`fprintf, fclose`).
- The alias may occur by assignment or function argument passing. Intraprocedural alias of resources is frequent but interprocedural alias of resources via argument passing is not frequent.
- Resource allocation rarely appears within loops. Even if it appears, every resource allocated in the loop should be deallocated or should have same specification as we can see in the following example:

```
while(?) {
  Lock* l = malloc(sizeof(Lock));
  lock(l); ... unlock(l);
}
```

- Values to identify paths are often constant limited to some simple integers.

Based on the observations above, we derived main abstraction principles for our type system as follows:

- Types are modeled as lattice elements or sets of lattice elements. We do not abstract resource related components of dynamic semantics but use set union ($\cup$) at the join point of analysis. We abstract other things if we know that they are not related to resource behavior and use normal join ($\sqcup$) at the join point. Only resource related functions are typed in the path sensitive way.
- Pointer variables that represent resources are translated to a normal variable in our simplified imperative language.
- We trace the alias information in the path sensitive way within function body under the assumption of no interprocedural alias introduced by function arguments.
- Resources are identified by allocation points. All resources allocated in the same program point should satisfy the same resource usage specification.
- Values that possibly identify paths are traced with constant propagation lattices. (for simpler presentation, we use sign domain in this paper)

3 A Language of Resource Usage: *RL*

3.1 Syntax

The syntax of resource usage language (*RL*) used to formalize our idea is given in Figure 2. A program is a sequence of function declarations which ends with an expression meaning the main function in C. All expressions are in the K-normal form[29,19,22]: every non-value expression is bound by a variable by `let`. A

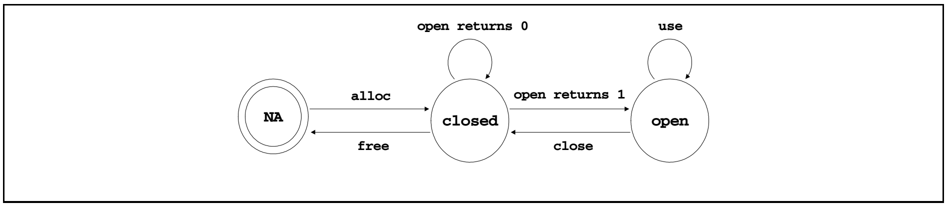


Fig. 1. FSM specification for File usage

value expression *val* is denoted as a pair of resource identifier and an integer. A *ResourceId* r_i denotes a resource allocated by allocation expression $\texttt{alloc}_i$ where i means the program point of the resource allocation. r_0 is used to express that it is not a resource. For example, an integer n in C is translated to a value (r_0, n) in RL. Note that $\texttt{alloc}_i$ is the only way to make a value (r_i, n) where $i \neq 0$. Variables in C including the resource pointer are directly translated to the variables in RL. Function application, assignment and `if` expression are standard language constructors. `let` expression is used for local variable declaration and sequencing. $\texttt{alloc}_i$, $\texttt{open}\,x$ and $\texttt{close}\,x$ are resource usage related expressions. In fact, they correspond to API functions in C which are related to resource usage. For example, the valid usage of file resources in C is summarized as in Figure 1. In our type system, this is represented in the notion of function type[1] as in Figure 6. Some of these resource related expressions are chosen as language constructs to present our idea clearly. $\texttt{use}\,x$ and $\texttt{free}\,x$ are omitted for presentation brevity. Loop, mutual recursion and global variables are excluded since they can be translated into RL. We do not consider structure and pointer arithmetic yet.

3.2 Dynamic Semantics

The dynamic semantics, which is specified in Figure 2, is given by a deterministic rewriting system $C \rightarrow C'$ mapping a configuration (a program state) to a new configuration. A *configuration* C consists of a tuple (γ, h, p, K) for program, (γ, h, e, K) for expression or $(finished)$. $(finished)$ means the termination of execution. An *environment* γ is a finite mapping of *Id* to *closure* or *value* which represents stack-like memory. A *resource heap* h is a finite mapping of resource id r_i to resource state q, where r_i means a resource allocated at the program point i. A resource heap can be understood as sub-part of the heap-like memory in C that contains resources in which we are interested. A *continuation* K is a sequence of (x, e) or γ which acts as a stack of next computation e or a stack of environment to be recovered after a function call.

Rewriting rules for program, value, variable, function application, assignment, `let`[2] and branch are given in the standard manner. The dynamic allocation semantics of resources is defined based on the observations in Section 2.

[1] The function type will be explained in Section 4.

[2] We assume let expressions always introduce a new variable to be free from scoping issue for brevity.

$p \in \mathit{Program}$	$::=$	$\texttt{fun}\, f(\vec{x})\!=\!e\,;p \mid e$	
$e \in \mathit{Expression}$	$::=$	val	value
	$\mid$	x	variable
	$\mid$	$f\,\vec{x}$	function app
	$\mid$	$x := x'$	assignment
	$\mid$	$\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}$	let (seq)
	$\mid$	$\texttt{if}\ \mathit{cond}\ \texttt{then}\ e_1\ \texttt{else}\ e_2$	branch
	$\mid$	$\texttt{alloc}_i$	resource allocation
	$\mid$	$\texttt{open}\ x$	resource open
	$\mid$	$\texttt{close}\ x$	resource close
$\mathit{cond} \in \mathit{Condition}$	$::=$	$? \mid x \le n \mid n \le x$	condition
$\mathit{val} \in \mathit{Value}$	$::=$	(Rid, n)	value
$\mathit{Rid} \in \mathit{ResourceId}$	$::=$	r_i	resource identifier
$x, f \in \mathit{Id}$			identifier
$\mathit{clos} \in \mathit{Closure}$	$=$	$\mathit{Env} \times \mathit{Id} \times \vec{\mathit{Id}} \times \mathit{Expression}$	closure
$q \in \mathit{Resource\_State}$	$::=$	$\mathit{open} \mid \mathit{closed}$	resource state
$\gamma \in \mathit{Env}$	$=$	$\mathit{Id} \overset{fin}{\to} (\mathit{Closure} \cup \mathit{Value})$	environment
$h \in \mathit{Resouce\_Heap}$	$=$	$\mathit{Rid} \overset{fin}{\to} \mathit{Resource\_State}$	resource heap
$K \in \mathit{Continuation}$	$::=$	$\epsilon \mid (x, e) :: K \mid \gamma :: K$	continuation
$C \in \mathit{Configuration}$	$:=$	$(\gamma, h, p, K) \mid (\gamma, h, e, K) \mid (\mathit{finished})$	configuration

[prog] $(\gamma, h, \texttt{fun}\, f(\vec{x})\!=\!e\,;p, K) \to (\gamma[f \mapsto \langle \gamma, f, \vec{x}, e\rangle], h, p, K)$

[var] $(\gamma, h, x, K) \to (\gamma, h, \gamma(x), K)$

[val-end] $(\gamma, h, \mathit{val}, \epsilon) \to (\mathit{finished})$ if $\forall r_i \in \mathit{Dom}(h). h(r_i) = \mathit{closed}$

[val-seq] $(\gamma, h, \mathit{val}, (x, e) :: K) \to (\gamma[x \mapsto \mathit{val}], h, e, K)$

[val-env] $(\gamma, h, \mathit{val}, \gamma' :: K) \to (\gamma', h, \mathit{val}, K)$

[assign] $(\gamma, h, x := x', K) \to (\gamma[x \mapsto \gamma(x')], h, (r_0, 0), K)$

[let] $(\gamma, h, \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}, K) \to (\gamma, h, e_1, (x, e_2) :: K)$

[app] $(\gamma, h, f\,\vec{y}, K) \to (\gamma'[f \mapsto \langle \gamma', f, \vec{x}, e\rangle][\vec{x} \mapsto \gamma(\vec{y})], h, e, \gamma :: K)$
if $\gamma(f) = \langle \gamma', f, \vec{x}, e\rangle$

[ifT] $(\gamma, h, \texttt{if}\ \mathit{cond}\ \texttt{then}\ e_1\ \texttt{else}\ e_2, K) \to (\gamma, h, e_1, K)$ if $(\gamma, \mathit{cond}) = \texttt{true}$

[ifF] $(\gamma, h, \texttt{if}\ \mathit{cond}\ \texttt{then}\ e_1\ \texttt{else}\ e_2, K) \to (\gamma, h, e_2, K)$ if $(\gamma, \mathit{cond}) = \texttt{false}$

[alloc] $(\gamma, h, \texttt{alloc}_i, K) \to (\gamma, h[r_i \mapsto \mathit{closed}], (r_i, 0), K)$
if $(r_i \notin \mathit{Dom}(h)) \vee (r_i \in \mathit{Dom}(h) \wedge h(r_i) = \mathit{closed})$

[openS] $(\gamma, h, \texttt{open}\ x, K) \to (\gamma, h[r_i \mapsto \mathit{open}], (r_i, 1), K)$
if $\gamma(x) = (r_i, n) \wedge i \neq 0 \wedge h(r_i) = \mathit{closed} \wedge$ open succeed

[openF] $(\gamma, h, \texttt{open}\ x, K) \to (\gamma, h, (r_i, 0), K)$
if $\gamma(x) = (r_i, n) \wedge i \neq 0 \wedge h(r_i) = \mathit{closed} \wedge$ open fail

[close] $(\gamma, h, \texttt{close}\ x, K) \to (\gamma, h[r_i \mapsto \mathit{closed}], (r_0, 0), K)$
if $\gamma(x) = (r_i, n) \wedge i \neq 0 \wedge h(r_i) = \mathit{open}$

[cond]
$(\gamma, x \le n) =$ if $\gamma(x).1 \le n$ then $\texttt{true}$ else $\texttt{false}$
$(\gamma, n \le x) =$ if $n \le \gamma(x).1$ then $\texttt{true}$ else $\texttt{false}$
$(\gamma, ?) = \texttt{true}$ or $\texttt{false}$

Fig. 2. Syntax and Dynamic Semantics of *RL*

`alloc`$_i$ first checks whether the allocation for the resource r_i has been performed already. If r_i has not been allocated before, it is allocated by mapping the state of resource as initial state. If r_i has been allocated already, `alloc`$_i$ checks whether the state of resource r_i is in the final state, and updates it to initial state in the FSM. `open` x first checks whether the state of the resource bound to x is *closed*. If `open`x succeeds, $(\gamma(x), 1)$ is returned. Otherwise, $(\gamma(x), 0)$ is returned. `close` x is defined similarly.

4 Type System of Resource Language

We formalize our analysis as a subtype system[25]. Intuitively, each *type* in our type system can be understood as a lattice element instrumented with type variables. Each lattice element (type) abstracts corresponding component of the dynamic semantics based on the abstraction principles discussed in Section 2, which is given in Figure 3.

4.1 Types

Before we proceed, we first define some notations and operators used in our type system. τ denotes an arbitrary type of our type system. X denotes an arbitrary syntactic element of our language. $\widehat{X}$ denotes a simple type of X. For example, $\widehat{1}$ is P (Plus) in our type system. $FTV(\tau)$ denotes the free type variables in type τ which is defined in standard manner on each type. We write $X.(n-1)$ to denote n_{th} element of a *tuple* type X. $\vec{X}$ is used to abbreviate various enumerations. For example, $\alpha_1, \cdots, \alpha_n$ is abbreviated to $\vec{\alpha}$. $\Gamma(\vec{x}) = (\vec{R}, \vec{D})$ is abbreviation for $\Gamma(x_1) = (R_1, D_1)\ \cdots\ \Gamma(x_n) = (R_n, D_n)$. $[\vec{x} \mapsto \vec{v}]$ is abbreviation for $[x_1 \mapsto v_1, \cdots, x_n \mapsto v_n]$. $[\vec{D}/\vec{\alpha}]$ is abbreviation for $[D_1/\alpha_1, \cdots, D_n/\alpha_n]$. A *substitution*

tv	$\in$ *TypeVariable*	$::=$	$\alpha \mid \rho \mid \beta$
$bool$	$\in$ *boolean*	$::=$	$t \mid f \mid tf$
d	$\in$ *sign*	$::=$	$\bot \mid \mathsf{M} \mid \mathsf{P} \mid \mathsf{Z} \mid \mathsf{MZ} \mid \mathsf{MP} \mid \mathsf{ZP} \mid \top$
D	$\in$ *Sign*	$::=$	$\alpha \mid \alpha \sqcup D \mid d$
R	$\in$ *ResourceId*	$::=$	$\rho \mid r_i$
v	$\in$ *ValueType*	$::=$	(R, D)
rs	$\in$ *ResourceState*	$::=$	$\bot \mid \mathsf{O} \mid \mathsf{C} \mid \top$
al	$\in$ *AllocState*	$::=$	$\bot \mid \mathsf{AL} \mid \mathsf{NA} \mid \top$
as	$\in$ *Assumption*	$::=$	$D \sqsubseteq D \mid R \doteq R \mid H \sqsubseteq H \mid D \# D \mid R \# R$
A	$\in$ *AssumptionSet*	$::=$	$\{as_1, \cdots, as_n\}$
ret	$\in$ *ReturnSet*	$::=$	$\{(v_1, H_1), ..., (v_n, H_n)\}$
ts	$\in$ *TypeScheme*	$::=$	$\forall \vec{\alpha}, \vec{\rho}, \beta.\{(A_1, ret_1), \cdots, (A_n, ret_n)\}$
Γ	$\in$ *TypeEnv*	$=$	$Id \rightarrow TypeScheme + ValueType$
h	$\in$ *ConstrainedHeap*	$::=$	$\{(R_1, al_1, rs_1), \cdots, (R_n, al_n, rs_n)\}$
H	$\in$ *ResourceHeap*	$::=$	$h \mid H \cdot [R \mapsto (al, rs)] \mid \beta$
Ω	$\in$ *ValueStatePairs*	$::=$	$\{(v_1, \Gamma_1, H_1), \cdots, (v_n, \Gamma_n, H_n)\}$

Fig. 3. Types

$$\boxed{Bas} = \left\{ \begin{array}{l} \bot \sqsubseteq \mathsf{C}, \bot \sqsubseteq \mathsf{O}, \mathsf{C} \sqsubseteq \top, \mathsf{O} \sqsubseteq \top,\ r_1 \doteq r_1, r_1 \# r_2, \cdots \\ \bot \sqsubseteq \mathsf{M}, \bot \sqsubseteq \mathsf{Z}, \bot \sqsubseteq \mathsf{P}, \bot \sqsubseteq \mathsf{MZ}, \mathsf{P} \sqsubseteq \mathsf{ZP}, \cdots, \\ \bot \sqsubseteq \mathsf{AL}, \bot \sqsubseteq \mathsf{NA}, \mathsf{AL} \sqsubseteq \top, \mathsf{NA} \sqsubseteq \top \end{array} \right\}$$

[hypoth] $$A \vdash as \quad \text{if}\,(as \in A) \text{ or } (as \in Bas)$$

[v$\sqsubseteq$] $$\frac{A \vdash R_1 \doteq R_2 \qquad A \vdash D_1 \sqsubseteq D_2}{A \vdash (R_1, D_1) \sqsubseteq (R_2, D_2)}$$

[$H_1\sqsubseteq$] $$\frac{\begin{array}{c} H_1 = \{(\vec{R}, \vec{al}, \vec{rs})\} \quad H_2 = \{(\vec{R'}, \vec{al'}, \vec{rs'})\} \\ \forall (R', al', rs') \in H_2. \exists (R, al, rs) \in H_1. \\ (A \vdash R \doteq R' \ \wedge\ A \vdash al \sqsubseteq al' \ \wedge\ A \vdash rs \sqsubseteq rs') \end{array}}{A \vdash H_1 \sqsubseteq H_2}$$

[$H_2\sqsubseteq$] $$\frac{\begin{array}{c} H_1 = H \cdot [R \mapsto (al, rs)] \quad H_2 = \{(\vec{R}, \vec{al}, \vec{rs})\} \\ \forall (R_i, al_i, rs_i) \in H_2. \\ (A \vdash R \doteq R_i \ \wedge\ A \vdash al \sqsubseteq al_i \ \wedge\ A \vdash rs \sqsubseteq rs_i) \\ \vee\ (A \vdash R \# R_i \ \wedge\ A \vdash H \sqsubseteq \{(R_i, al_i, rs_i)\}) \end{array}}{A \vdash H_1 \sqsubseteq H_2}$$

[A] $$\frac{\forall as \in A'. A \ \vdash\ as}{A \ \vdash\ A'}$$

Fig. 4. Constraint rules: ordering, equality, disjointness

is a set of simultaneous replacements for type variables denoted as $[\vec{\tau}/\vec{tv}]$ where tv_i's are distinct. We write the application of a substitution S to type τ as τS, and we write the composition of substitution S and S' as SS'.

Integer values are abstracted into d of *sign* domain. A *Sign* type D is instrumented sign domain with type variable α. The type variable α means an integer value given as function argument. A *ResourceId* type R is defined without any abstraction because we do not want to lose any information about resources. So, we do not introduce join instrumentation for ρ in contrast to the *Sign* type. A *ResourceState* type rs is a lattice domain generated by lifting all the states of FSM specification with $\bot, \top$ like constant propagation domain. An *AllocState* type al is a lattice domain that denotes whether a resource is allocated (AL) or not (NA).

The *ResourceHeap* type H abstracts *Resource_Heap* of the dynamic semantics. A resource heap $\emptyset$ is concretized to every concrete heap possible (i.e., it is the *top* heap). $\{(r_1, \mathsf{AL}, \mathsf{O})\}$ is concretized to every concrete heap h where resource r_1 is allocated and the state is in *open*, and so on. A *bottom* heap can be understood as $\{(r_i, \bot, \bot) \mid r_i \in$ all possible $r_i\}$. Because we will use the heap in the context of constraints, we need an additional representation to deal with *update* on a heap. $H \cdot [R \mapsto (al, rs)]$ means a heap H where the most recent update is R to (al, rs). We name the right part of $\cdot$ as *history* of the heap.

A *TypeEnv* Γ is a finite mapping of $Id \mapsto \tau$. A map in Γ is either $f \mapsto ts$ denoting the *TypeScheme* of function f or $x \mapsto v$ denoting the *ValueType* of variable x. We write $\Gamma(Id)$ to denote the type τ bound to Id in Γ. We write $\Gamma[Id \mapsto \tau]$ to denote the environment Γ' after updating a map $Id \mapsto \tau$. A

TypeScheme ts is defined as a set of input, output relations which is quantified by three kinds of type variables. (ρ_i,α_i) means the i_{th} argument of a function and β means input resource heap. A *ReturnSet* ret_i is a set of tuple (v_j, H_j) which means the set of possible output (path) with respect to the input (path) constrained by A_i. Note our typing rules manipulate A_i to be always disjoint assumption set for a ts. We write $A \vdash ts_1 \succ ts_2$ denote instantiation of ts_1 with respect to assumptions A. ts_2 is a type scheme after renaming all type variables of ts_1 to be different from $FTV(A)$ (formally defined in our technical memo[18]).

There are three kinds of *Assumption* (*assertion*, *constraint*) *as* that we are interested in proving: ordering (inclusion) ($\sqsubseteq$), equality ($\doteq$) and disjointness ($\#$). Intuitively, an assumption $\alpha_i \sqsubseteq \mathsf{P}$ denotes a path that the i_{th} argument of a function is given as P. These assertions will in general depend on a set of assumption (*assumptions*) A, which contains the typings of basic ordering, equality, disjointness of our lattice based type. So the basic judgements of our type system are:

$$A \vdash_{as} as \quad \text{and} \quad A \vdash_A A'$$

Judgement $A \vdash_{as} as$ is read as "Under assumptions A, as is typed". Simply, "A types as". Judgement $A \vdash_A A'$ types set of assumptions. Typing rules for these judgements[3] are given in Figure 4. Typing rules for each assertion is defined in the standard context of the lattice. Note that the heap ordering is defined to be consistent with the abstraction explained in the heap type. Full assumption typing rules (e.g. $\Omega_{\sqsubseteq}$), which is required to explain the computation of fixpoint in inference algorithm of Section 5, is given in [18].

4.2 Typing Rules

Main judgements of our type system are:

$$(1)\ A, \Gamma, H \vdash e : \Omega \qquad (2)\ A, \Gamma, H \vdash p : \Omega \qquad (3)\ A, \Gamma \vdash cond : bool$$

(1) is read as "Under assumptions A, type environment Γ and resource heap H, the expression e has type Ω". Intuitively, it is read as "Ω is the all possible output result (path) of e for an input (path) (A, Γ, H)". (2) is read similarly. (3) is read as "Under assumptions A and type environment Γ, *cond* has type *bool*".

Typing rule for *Program* is given in Figure 5. To type a function f, we first prepare a set of well partitioned assumptions $A_1, \cdots, A_n$ which denote a set of disjoint input paths[4]. Each A_i can be understood as assumptions (constraints) on each (ρ_i, α_i) in Γ_f and β. Then, we prepare the function environment Γ_f and a resource heap β constrained by each A_i. If each input path (A_i, Γ_f, β) types e as Ω_i, the type of the function f is $\forall \vec{\alpha}, \vec{\rho}, \beta.\{(A_1, ret_1), \cdots, (A_n, ret_n)\}$. Note that Γ_f used to type e contains its own function type in addition to the arguments types. Finally, we type the remaining part p of the program using the type scheme proved to be the type of function f.

[3] We abbreviate the type of judgements like $\vdash_{as}$, $\vdash_A$ freely in this paper for simplicity. Symbols like $(\bot)$ and operators like FTV is overloaded freely in similar way.

[4] Formal definition of well partitioned assumptions $\vdash (A_1, \cdots, A_n)$ and well-formedness $A \vdash (\Gamma, H)$ are given in [18].

[Program]
$$\frac{\begin{array}{c}\Gamma_f = \Gamma[f \mapsto \forall\vec{\alpha},\vec{\rho},\beta.\{(A_1, ret_1),\cdots,(A_n, ret_n)\}][\vec{x} \mapsto (\vec{\rho},\vec{\alpha})] \\ \vdash (A_1,\cdots,A_n) \quad A_1 \vdash (\Gamma_f,\beta) \quad \cdots \quad A_n \vdash (\Gamma_f,\beta) \\ A_1,\Gamma_f,\beta \vdash e : \Omega_1 \qquad \cdots \qquad A_n,\Gamma_f,\beta \vdash e : \Omega_n \\ ret_i = \{(v_j, H_j) \mid (v_j,\Gamma_j,H_j) \in \Omega_i\} \\ A,\Gamma[f \mapsto \forall\vec{\alpha},\vec{\rho},\beta.\{(A_1,ret_1),\cdots,(A_n,ret_n)\}], H \vdash p : \Omega\end{array}}{A,\Gamma,H \vdash \texttt{fun}\, f(\vec{x}){=}e\ ;p : \Omega}$$

[App]
$$\frac{\begin{array}{c}A \vdash \Gamma(f) \succ \forall\vec{\alpha},\vec{\rho},\beta.\{(A_1,ret_1),\cdots,(A_n,ret_n)\} \\ \Gamma(\vec{x}) = (\vec{R},\vec{D}) \qquad S = [\vec{D}/\vec{\alpha}][\vec{R}/\vec{\rho}][H/\beta] \qquad A \vdash A_i S \\ \Omega = \{(v_{ik}S, \Gamma, (H_{ik}S)) \mid (v_{ik}, H_{ik}) \in ret_i\}\end{array}}{A,\Gamma,H \vdash f\ \vec{x} : \Omega}$$

[Value]
$$A,\Gamma,H \vdash val : \{(\widehat{val},\Gamma,H)\}$$

[Var]
$$A,\Gamma,H \vdash x : \{(\Gamma(x),\Gamma,H)\}$$

[Assign]
$$A,\Gamma,H \vdash x := x' : \{((r_0,\mathsf{Z}),\Gamma[x \mapsto \Gamma(x')],H)\}$$

[Let]
$$\frac{\begin{array}{c}A,\Gamma,H \vdash e_1 : \{(v_1,\Gamma_1,H_1),\cdots,(v_n,\Gamma_n,H_n)\} \\ A,\Gamma_1[x \mapsto v_1],H_1 \vdash e_2 : \Omega_{21} \\ \cdots \\ A,\Gamma_n[x \mapsto v_n],H_n \vdash e_2 : \Omega_{2n}\end{array}}{A,\Gamma,H \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \Omega_{21} \uplus \cdots \uplus \Omega_{2n}}$$

[IfT]
$$\frac{A,\Gamma \vdash cond : t \quad A,\Gamma,H \vdash e_1 : \Omega_1}{A,\Gamma,H \vdash \texttt{if}\ cond\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \Omega_1}$$

[IfF]
$$\frac{A,\Gamma \vdash cond : f \quad A,\Gamma,H \vdash e_2 : \Omega_2}{A,\Gamma,H \vdash \texttt{if}\ cond\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \Omega_2}$$

[IfTF]
$$\frac{A,\Gamma \vdash cond : tf \qquad A,\Gamma,H \vdash e_1 : \Omega_1 \qquad A,\Gamma,H \vdash e_2 : \Omega_2}{A,\Gamma,H \vdash \texttt{if}\ cond\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \Omega_1 \uplus \Omega_2}$$

[AllocA]
$$\frac{A \vdash H \sqsubseteq \{(r_i,\mathsf{NA},\top)\}}{A,\Gamma,H \vdash \texttt{alloc}_i : \{((r_i,\mathsf{Z}),\Gamma,H \cdot [r_i \mapsto (\mathsf{AL},\mathsf{C})])\}}$$

[AllocB]
$$\frac{A \vdash H \sqsubseteq \{(r_i,\mathsf{AL},\mathsf{C})\}}{A,\Gamma,H \vdash \texttt{alloc}_i : \{((r_i,\mathsf{Z}),\Gamma,H \cdot [r_i \mapsto (\mathsf{AL},\mathsf{C})])\}}$$

[Open]
$$\frac{\Gamma(x) = (R,D) \quad A \vdash R\#r_0 \quad A \vdash H \sqsubseteq \{(R,\mathsf{AL},\mathsf{C})\}}{A,\Gamma,H \vdash \texttt{open}\ x : \{((R,\mathsf{P}),\Gamma,H \cdot [R \mapsto (\mathsf{AL},\mathsf{O})]),((R,\mathsf{Z}),\Gamma,H)\}}$$

[Close]
$$\frac{\Gamma(x) = (R,D) \quad A \vdash R\#r_0 \quad A \vdash H \sqsubseteq \{(R,\mathsf{AL},\mathsf{O})\}}{A,\Gamma,H \vdash \texttt{close}\ x : \{((r_0,\mathsf{Z}),\Gamma,H \cdot [R \mapsto (\mathsf{AL},\mathsf{C})])\}}$$

Fig. 5. Typing Rules: *Program*, *Expression*

$$\begin{array}{ll}
\texttt{alloc}_i & : \forall\beta.\,\{\beta \sqsubseteq \{(r_i, \mathsf{NA}, \top)\}\} \rightarrow \{((r_i, \mathsf{Z}), \beta \cdot [r_i \mapsto (\mathsf{AL}, \mathsf{C})])\} \\
 & \quad \{\beta \sqsubseteq \{(r_i, \mathsf{AL}, \mathsf{C}))\}\} \rightarrow \{((r_i, \mathsf{Z}), \beta)\} \\
\texttt{open}\, x & : \forall\alpha, \rho, \beta.\{\rho\# r_0, \beta \sqsubseteq \{(\rho, \mathsf{AL}, \mathsf{C})\}\} \rightarrow \{((\rho, \mathsf{P}), \beta \cdot [\rho \mapsto (\mathsf{AL}, \mathsf{O})]),\ ((\rho, \mathsf{Z}), \beta)\} \\
\texttt{close}\, x & : \forall\alpha, \rho, \beta.\{\rho\# r_0, \beta \sqsubseteq \{(\rho, \mathsf{AL}, \mathsf{O})\}\} \rightarrow \{((r_0, \mathsf{Z}), \beta \cdot [\rho \mapsto (\mathsf{AL}, \mathsf{C})])\} \\
\texttt{use}\, x & : \forall\alpha, \rho, \beta.\{\rho\# r_0, \beta \sqsubseteq \{(\rho, \mathsf{AL}, \mathsf{O})\}\} \rightarrow \{((r_0, \mathsf{Z}), \beta)\} \\
\texttt{free}\, x & : \forall\alpha, \rho, \beta.\{\rho\# r_0, \beta \sqsubseteq \{(\rho, \mathsf{AL}, \mathsf{C})\}\} \rightarrow \{((r_0, \mathsf{Z}), \beta \cdot [\rho \mapsto (\mathsf{NA}, \mathsf{C})])\}
\end{array}$$

Fig. 6. Type specification for File resource

Typing rule [App] for application $f\,\vec{x}$ is defined as follows: we first prepare the calling context of function f as a substitution S which denotes arguments and heap passing. Then we find which input path constrained by A_iS of function f is typable under the current assumptions A. If there is a typing (exists an input assumptions compatible with the calling context), result type (paths) is generated by replacing type variables of ret_i to the current calling context.

Typing rules [Value], [Var], [Assign] are defined in the standard way. For [Let], we type e_1 first under the given input path, and then we type e_2 under each output path from e_1. $\uplus$ is defined as normal join($\sqcup$) or set union($\cup$). In general, choosing $\sqcup$ or $\cup$ controls the level of path sensitivity in our type system. For our analysis, distinguishing paths is meaningful only when resource related part of program state is different. For example, with (Γ_1, H_1) and (Γ_2, H_2), if Γ_1, Γ_2 has the same location binding (same alias state) and $H_1 = H_2$, then we join($\sqcup$) two paths. Otherwise we union ($\cup$) two paths. The concrete algorithm for $\uplus$ is given in Section 5. [IfT], [IfF], [IfTF] rules are path sensitive extension of standard `if` typing. If we can conclude that the *cond* is true under the given input path (A, Γ), type of `if` expression is the type of e_1. False case is typed similarly. If we cannot conclude any of them, type of `if` expression is $\Omega_1 \uplus \Omega_2$. Typing rules for *cond* is given in [18].

There are two rules [AllocA], [AllocB] for $\texttt{alloc}_i$. If H is a heap where r_i is not allocated, denoted as $A \vdash H \sqsubseteq \{(r_i, \mathsf{NA}, \top)\}$, we update the heap such that r_i is allocated and the resource state is initial. If H is a heap where r_i is allocated, we check that the current state of r_i is the final state denoted as $A \vdash H \sqsubseteq \{(r_i, \mathsf{AL}, \mathsf{C})\}$. [Open] rule is defined as follows: we first check the value bound to x is really a resource by using $A \vdash R\# r_0$, then we check whether H is a heap where R is allocated and the state is C. Note that $\texttt{open}\, x$ generates two output paths identifiable by the *Sign* value (either P or Z). [Close] rule is defined similarly. Note that [AllocA], [AllocB], [Open] and [Close] ([Use], [Free]) can be equally specified by function types as in Figure 6, meaning the specification and verification is not restricted to the file resource usage.

4.3 Correctness of Type System

Theorem 1 [Correctness of type system]. *If a Configuration C is typed, then C is ($finished$) or it goes without type error.*

Proof: The complete proof is in [18].

5 Type Inference

The primary goal of type inference is to discover a minimal well-formed partition of a given assumption set, which enables us to get a path sensitive type with respect to the resource usage. Suppose that a function f takes n arguments, and that m among n arguments are of resource types. When the function f uses k resources in the body, the maximum size of a well-formed partition is $m^{(k+1)} \cdot 3^{(n+k)}$. We infer the type of a function with the environment that maps each argument x_i to (ρ_i, α_i). For each ρ_i, there are at most $(k+1)$ valid assumptions: $\rho_i \doteq r_0, \cdots, \rho_i \doteq r_k$, and $(\rho_i \# r_0 \wedge \cdots \wedge \rho_i \# r_k)$ where each r_i denotes one of k resources. For each α_i, there are three disjoint assumptions: $\alpha_i \sqsubseteq \mathsf{M}$, $\alpha_i \sqsubseteq \mathsf{Z}$, and $\alpha_i \sqsubseteq \mathsf{P}$. For each resource r_i, we consider the following three cases: $(r_i, \mathsf{NA}, \top)$, $(r_i, \mathsf{AL}, \mathsf{C})$, and $(r_i, \mathsf{AL}, \mathsf{O})$.

In this paper, we design a preliminary type inference algorithm based on lazy partitioning. Our algorithm automatically partitions the given assumption set by augmenting it with stronger constraints on the value of an argument α or the heap β, whenever the stronger assumptions help more precise typing. For the resource binding ρ of an argument, we assume the maximal partition $A_1, \cdots, A_n$, where each A_i contains different assumptions on ρ. We call this *the initial partition*. For example, with a resource-type argument ρ and two resources r_1, r_2, the initial partition is comprised of $\{(\rho \# r_0), (\rho \doteq r_1), (\rho \# r_2)\}$, $\{(\rho \# r_0), (\rho \# r_1), (\rho \doteq r_2)\}$, and $\{(\rho \# r_0), (\rho \# r_1), (\rho \# r_2)\}$. We run the algorithm on each assumption set in the initial partition and combine the results. We conjecture the size of the initial partition is small enough, based on the observations in Section 2. Furthermore, since we prohibit aliases between arguments, the number of valid assumption sets is much less than $m^{(k+1)}$.

5.1 Type Representations and Operations

In the algorithm, we always use a normalized assumption set that satisfies the following conditions: (1) for each variable α, there is exactly one constraint on α of the form $\alpha \sqsubseteq d$, (2) there is exactly one heap constraint of the form $\beta \sqsubseteq H$ where H is a sorted list of (R, al, rs), and (3) H mentions all the resources in the analysis scope[5]; when there are two resources R_1 and R_2, $\{(R_1, al, rs)\}$ is extended to $\{(R_1, al, rs), (R_2, \top, \top)\}$.

We define two operations on assumption sets for the algorithm: the satisfiability check and the conjunction of assumption sets. An assumption set is not satisfiable if it contains $\alpha \sqsubseteq \bot$ or $\beta \sqsubseteq H$, where $(R, \bot, \_) \in H$ or $(R, \_, \bot) \in H$. The conjunction $A_1 \sqcap A_2$ of two assumption sets is defined as the following: $A_1 \sqcap A_2 = \{\vec{\alpha} \sqsubseteq (D_1 \sqcap D_2) \,|\, \vec{\alpha} \sqsubseteq D_i \in A_i\} \cup \{\beta \sqsubseteq (H_1 \sqcap H_2) \,|\, \beta \sqsubseteq H_i \in A_i\} \cup \{\rho \doteq R \mid \rho \doteq R \in A_i\} \cup \{\rho \# R \mid \rho \# R \in A_i\}$. With normalized assumption sets, these operations can be performed in an efficient way. The satisfiability check requires

[5] An analysis scope specifies a module (i.e. a set of mutually recursive functions), which should be analyzed at the same time.

```
$I_p(A, \Gamma, H, \texttt{fun}\, f(\vec{x}) = e\ ;p) =$
    let $F = \text{fix}\, \lambda\, \text{F}.$
        let $\Gamma_f = \Gamma[f \mapsto \forall\vec{\alpha}, \vec{\rho}, \beta.\ F][\vec{x} \mapsto (\vec{\alpha}, \vec{\rho})]$
            $X = \forall(A_i, \_) \in F.\ \cup\ I_e(A_i, \Gamma_f, \beta,\ e)$
        in $\{(A_i, \{(v_j, H_j) \mid (v_j, \_, H_j) \in \Omega_i\}) \mid (A_i, \Omega_i) \in X\}$ end
    in $I_p(A, \Gamma[f \mapsto \forall\vec{\alpha}, \vec{\rho}, \beta.\ F], H, p)$ end
$I_p(A, \Gamma, H, e) = I_e(A, \Gamma, H, e)$

$I_e(A, \Gamma, H, v) = \{(A, \{(\hat{v}, \Gamma, H)\})\}$
$I_e(A, \Gamma, H, x) = \{(A, \{(\Gamma(x), \Gamma, H)\})\}$
$I_e(A, \Gamma, H, f\ \vec{x}) =$
    assume $\Gamma(f) = \forall\vec{\alpha}, \vec{\rho}, \beta.\{(A_1, ret_1), \cdots, (A_n, ret_n)\}$
    assume $\Gamma(x_i) = (\vec{R}, \vec{D})$
    let    $S = [\vec{D}/\vec{\alpha}][\vec{R}/\vec{\rho}][H/\beta]$
           $\Omega_i = \{(v_j S, \Gamma, H_j S) \mid (v_j, H_j) \in ret_i\ \wedge\ \mathit{satisfiable}(A \sqcap \mathit{simplify}(A_i S))\}$
           $X = \{(A \sqcap \mathit{simplify}(A_i S), \Omega_i) \mid \Omega_i \neq \emptyset\}$
    in     $\mathit{assert}(X \neq \emptyset); X$ end
$I_e(A, \Gamma, H, x := x') = \{(A, \{((r_0, Z), \Gamma[x \mapsto \Gamma(x')], H)\})\}$
$I_e(A, \Gamma, H, \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}) =$
    let $\{(A_1, \Omega_1), \cdots, (A_n, \Omega_n)\} = I_e(A, \Gamma, H, e_1)$
        $X_i = I_e(A_i, \Gamma_1[x \mapsto v_1], H_1, e_2) \uplus \cdots \uplus I_e(A_i, \Gamma_m[x \mapsto v_m], H_m, e_2)$
          where $\Omega_i = \{(v_1, \Gamma_1, H_1), \cdots, (v_m, \Gamma_m, H_m)\}$
    in $X_1 \cup \cdots \cup X_n$ end
$I_e(A, \Gamma, H, \texttt{if}\ \mathit{cond}\ \texttt{then}\ e_1\ \texttt{else}\ e_2) =$
    let  $(A_t, A_f, A_u) = \mathit{partition}(\Gamma, \mathit{cond})$
         $(A'_t, A'_f, A'_u) = (\mathit{simplify}(A_t), \mathit{simplify}(A_f), \mathit{simplify}(A_u))$
    in   if $A \sqsubseteq A'_t$ then $I_e(A, \Gamma, H, e_1)$
         else if $A \sqsubseteq A'_f$ then $I_e(A, \Gamma, H, e_2)$
         else if $\mathit{profitable}(A, A'_t, A'_f, \Gamma, H, e_1, e_2)$ then
             let $X_t = \mathit{satisfiable}(A \sqcap A'_t)\ ?\ I_e(A \sqcap A'_t, \Gamma, H, e_1)\ :\ \emptyset$
                 $X_f = \mathit{satisfiable}(A \sqcap A'_f)\ ?\ I_e(A \sqcap A'_f, \Gamma, H, e_2)\ :\ \emptyset$
                 $X_u = \mathit{satisfiable}(A \sqcap A'_u)\ ?$
                   $I_e(A \sqcap A'_u, \Gamma, H, e_1) \uplus I_e(A \sqcap A'_u, \Gamma, H, e_2)\ :\ \emptyset$
             in $X_t \cup X_f \cup X_u$ end
         else $I_e(A, \Gamma, H, e_1) \uplus I_e(A, \Gamma, H, e_2)$
    end

(* For simplicity, assumption sets are presented not in the normal form *)
$\mathit{partition}(\Gamma, \mathit{cond})$ = case $\mathit{abstract}(\mathit{cond})$ of
  |        $? \rightarrow (\{\Gamma(x) \sqsubseteq \bot\},\ \{\Gamma(x) \sqsubseteq \bot\},\ \{\Gamma(x) \sqsubseteq \top\})$
  | $x \leq \mathsf{M} \rightarrow (\{\Gamma(x) \sqsubseteq \bot\},\ \{\Gamma(x) \sqsubseteq \mathsf{ZP}\},\ \{\Gamma(x) \sqsubseteq \mathsf{M}\})$
  |  $x \leq \mathsf{P} \rightarrow (\{\Gamma(x) \sqsubseteq \mathsf{MZ}\},\ \{\Gamma(x) \sqsubseteq \bot\},\ \{\Gamma(x) \sqsubseteq \mathsf{P}\})$
  |  $x \leq \mathsf{Z} \rightarrow (\{\Gamma(x) \sqsubseteq \mathsf{MZ}\},\ \{\Gamma(x) \sqsubseteq \mathsf{P}\},\ \{\Gamma(x) \sqsubseteq \bot\})$
  | $\mathsf{M} \leq x \rightarrow (\{\Gamma(x) \sqsubseteq \mathsf{ZP}\},\ \{\Gamma(x) \sqsubseteq \bot\},\ \{\Gamma(x) \sqsubseteq \mathsf{M}\})$
  |  $\mathsf{Z} \leq x \rightarrow (\{\Gamma(x) \sqsubseteq \mathsf{ZP}\},\ \{\Gamma(x) \sqsubseteq \mathsf{M}\},\ \{\Gamma(x) \sqsubseteq \bot\})$
  |  $\mathsf{P} \leq x \rightarrow (\{\Gamma(x) \sqsubseteq \bot\},\ \{\Gamma(x) \sqsubseteq \mathsf{MZ}\},\ \{\Gamma(x) \sqsubseteq \mathsf{P}\})$
```

Fig. 7. Type Inference Algorithm

$O(n + |H|)$ time and the conjunction of assumption sets requires $O(n)$ time, where n is the number of distinct variables.

We represent a heap by $\beta \cdot \mathit{history}$, where *history* is the record of all updates on the heap β. However, we are interested in only the final configuration of the heap, it is sufficient to keep the set of visible updates instead of all history. The set representation enables efficient manipulation of heap constraints. In complexity analysis, we assume a data structure for H allowing $O(1)$ accesses to (R, al, rs) using R as a key. Note that $O(|\mathit{history}|) = O(|H|) = O(k)$ where k is the number of resources in the analysis scope.

The function *simplify* transforms the heap constraint of the form $\beta \cdot \mathit{history} \sqsubseteq H$ into the form $\beta \sqsubseteq H'$ in $O(|H|)$ time by the following algorithm: for all $(R, al, rs) \in \mathit{history}$, (1) if $(R, al', rs') \in H$ satisfies $al \sqsubseteq al'$ and $rs \sqsubseteq rs'$, then replace $(R, al', rs') \in H$ with $(R, \top, \top)$. (2) Otherwise, $H' = \bot$. The second case indicates that the heap constraint is not satisfiable. For constraints with no variables, *simplify* checks if the constraints are satisfiable then drops it.

Finally, we describe our implementation of the path sensitive join $\uplus$:

```
$\Omega_1 \uplus \Omega_2 =$
    let $\Omega' = \Omega_1$
    in  for each $(v, \Gamma, H) \in \Omega_2$ do
            if $(v', \Gamma', H) \in \Omega' \wedge \mathit{SameResourceBinding}(\Gamma, \Gamma') \wedge (v.0 = v'.0)$
            then $\mathit{replace}\ (v, \Gamma, H)\ \mathit{in}\ \Omega'\ \mathit{with}\ (v \sqcup v', \Gamma \sqcup \Gamma', H)$
            else $\mathit{insert}\ (v', \Gamma', H)\ \mathit{to}\ \Omega'$;
        $\Omega'$
```

Under the constraint $\beta \sqsubseteq H$, we consider that two heaps $\beta \cdot \mathit{history}_1$ and $\beta \cdot \mathit{history}_2$ are identical if $H \cdot \mathit{history}_1 = H \cdot \mathit{history}_2$. We transform $H \cdot \mathit{history}_i$ into H_i of a set form by applying all the assignments in $\mathit{history}_i$ to H. Since a heap comparison takes $O(|H|)$ time, the join takes $O((|H| + |\Gamma|) \cdot |\Omega_1| \cdot |\Omega_2|)$.

5.2 The Algorithm

Our type inference algorithm is described in Figure 7. The structure of the algorithm is as follows:

$$\begin{array}{rl} (1)\ I_p(A, \Gamma, H, p) & : \{(A_1, \Omega_1), \cdots, (A_n, \Omega_n)\} \\ (2)\ I_e(A, \Gamma, H, e) & : \{(A_1, \Omega_1), \cdots, (A_m, \Omega_m)\} \end{array}$$

Beginning with the well-formed state $A \vdash (\Gamma, H)$, the algorithm produces a well-formed partition $A_1, \cdots, A_n$ of A, where each A_i preserves well-formedness of the state, *i.e.* $A_i \vdash (\Gamma, H)$. Each Ω_i is a correct type of e *w.r.t.* the type system in Section 4 under A_i,Γ, and H. Resource manipulation expressions such as `alloc`$_i$ or `open` x are regarded as function applications that have the types in Figure 6. Thus, Figure 7 does not have any rules for those expressions.

For function definitions, we require the fixpoint iteration, since our language permits recursive calls. We begin the iteration with the following type: $\{(A \cup A_\rho \cup \{\vec{\alpha} \sqsubseteq \top, \beta \sqsubseteq \top\}, \{\})\}$ where A is given and A_ρ is an element of the initial partition. Note that A should not have constraints on α and β, since α and β are bounded to the function type. The iteration always terminates because the domain is finite and the transfer function is monotonic; the function further partitions assumption sets, and Ω grows for each path.

According to the typing rule [App], the algorithm should produce A_i validating exactly one of the assumption sets in the type of the function to be called. Suppose the type of the callee is $\{(A'_1, \mathit{ret_1}), (A'_2, \mathit{ret_2})\}$. When the given assumption set A can validate neither A'_1 nor A'_2, we need to refine A to use the type of the callee. Since A'_1 and A'_2 are disjoint, $A \sqcap A'_1$ and $A \sqcap A'_2$ is a well-formed partition of A satisfying such requirement. Unsatisfiable assumption set denotes infeasible paths. The function *satisfiable* detects such assumption sets according to the rules in Section 5.1, and we discard them. Since substitutions may produce a constraint breaking the normal form condition, we use the function *simplify* for the transformation.

We recognize the resource states by values from resource manipulation expressions such as `open` x, since our language does not allow to identify the state of a resource directly. If the purpose of a branch expression is to identify a resource state, the algorithm should partition the given assumption set A by the branch condition. In the algorithm, the function *profitable* determines whether partitioning helps more precise typing or not. One safe implementation of *profitable* is always returning true. The path sensitive join $\uplus$ would recover partitions that do not help recognizing resource states. However, we plan to develop light-weight analyses or heuristics for early detection of useless partitioning, e.g. both branches do not touch resources. Note that the actual implementation of *profitable* does not affect the soundness of the algorithm. The function *partition* generates three disjoint assumption sets according to the branch condition; the first two is for true and false cases, and the last is for the indeterminable case. Again, $A \sqcap A_t$, $A \sqcap A_f$, and $A \sqcap A_u$ forms a well-formed partition, and we discard unsatisfiable ones. When the branch condition is indeterminate, the path sensitive join $\uplus$ combines the results of both branchs.

We extend the path sensitive join $\uplus$ to the results of I_e. Suppose two results $\{(A_1, \Omega_1), \cdots, (A_n, \Omega_n)\}$ and $\{(A'_1, \Omega'_1), \cdots, (A'_m, \Omega'_m)\}$. First, $\uplus$ constructs an intermediate result with the finest well-formed partition: $\{\cdots, (A_i \sqcap A'_j, \Omega_i \uplus \Omega'_j), \cdots\}$ where $A_i \sqcap A'_j$ is satisfiable. Then, $\uplus$ recovers the useless partitions in the result; $\uplus$ merges (A, Ω) and (A', Ω') into $(A \sqcup A', \Omega \uplus \Omega')$ when Ω and Ω' have the same resource state, i.e. the same resource heap and binding.

5.3 Correctness of the Algorithm

Theorem 2 [Correctness of the algorithm].
If $I_p(A, \Gamma, H, p) = \{(A_1, \Omega_1), \cdots, (A_n, \Omega_n)\}$*, then* $A_i, \Gamma, H \vdash p : \Omega_i$.
If $I_e(A, \Gamma, H, e) = \{(A_1, \Omega_1), \cdots, (A_n, \Omega_n)\}$*, then* $A_i, \Gamma, H \vdash e : \Omega_i$.

Proof: Induction on the structure of p and e. The complete proof is in [18].

6 Related and Future Works

Type systems for region-based memory managements[1,30] ensure that deallocated regions are no longer accessed. Type systems for race detection[11,12] ensure that appropriate locks will be acquired before a shared object is accessed. Type system for object initialization[15] ensures that every object is initialized before it is accessed. The type system for JVM of [21] ensures that an object that has been locked will be eventually unlocked. In contrast to these works, our type system is formalized in more general setting with path sensitivity.

Resource usage analysis[17,20], CQual[14] and type state verification[27,10] are formalized as general analyses to verify resource usage properties as our approach. Primary difference from our approach is that they do not formalize the ideas for identifying actions which is dependent on values of variables (path insensitive). There are other approaches which can verify resource usage protocols with the help of *active* cooperation of programmers[6,9,13,28]. For example, Vault[6] system keeps track of the state of each resource to control accesses to resources. In contrast, our work does not need user intervention.

SLAM[2], Blast[16], and other software model checking projects can analyze resource usage properties in cooperation with path information which is identifiable by the known values of variables as presented in this paper. The main difference from our approach is that these are whole program analyses. We can analyze each function separately whenever all used function summaries (not source code) are given. Summaries can be specified in the path sensitive way for unavailable program fragments.

ESP[5] is a path sensitive resource usage verification tool that is focusing on scalability. It analyzes programs in the path sensitive way only when two arms of a branch have different resource state, like $\uplus$ in our work. It verifies each resource separately based on the observation that property of each resource is usually independent. However, it is not clear whether this approach can scale up arbitrarily because the approach still uses whole program analysis. The precision and scalability of ESP can be largely dependent on the pointer analysis used[24,7]. We have a different point of view on scalability. First, we attack the scalability by formalizing our analysis as modular analysis; we summarize only once for each function. Second, rather than using a separate whole program alias analysis to trace resources, our analysis traces it directly in a path sensitive way together with the resource usage when it computes the summary of a function. In other words, interprocedural aliases are just rejected and intraprocedural aliases are traced in the path sensitive way. Experiments to show effectiveness of this idea are left as a future work.

In contrast to sound approaches mentioned above, there are several approaches that aim to detect bugs effectively at the cost of unsoundness[8,31,4,3]. MC[8] is a tool that aims to find resource usage related bugs with few false alarms. Even though it uses a number of heuristics to remove false alarms, it cannot fully compensate for the information lost due to the under-approximation. For example, MC may generate a false alarm for the following code:

```
if (x) { fp = fopen("file"); }
if (x && fp) { fclose(fp); }
```

Saturn[31] is a SAT solver based approach to detect bugs in the path sensitive and modular way. It translates a C function into an input language of a SAT solver which encodes the possible state and control of code. Then it tries to find all the satisfiable solutions to summarize the function in an exhaustive way exploiting the power of SAT solvers which is much improved recently[23]. Their experimental results show the effectiveness of SAT based approach. It also shows us the feasibility of exhaustive path sensitive summarization on this problem. One weakness they pointed out is the way of handling loops and recursion. Saturn unfolds a loop for some bounded number of times and then tries to find bugs. In this reason, it may generate false alarms for the following program fragment.

```
my_fopen() {  //extracted from 175.vpr of SPEC benchmark suite.
  while (1) {
     scanf("%s",fname);
     if ((fp = fopen(fname,flag)) != NULL) break;
  }
  return (fp);
}
```

Saturn can be understood as an effort to overcome the problems due to the under-approximation of the real properties for the purpose of error detection. To the contrary, our work is understood as an effort to overcome the problems due to the over-approximation for the purpose of verification. Our lazy path partitioning algorithm discussed in Section 5 is one potential strength of our work over Saturn's exhaustive summary computation.

Future works are summarized as follows: we first plan to show that our approach is effective by experiments with full implementation. Extensions for structures and pointers in C programs will be considered as well.

7 Conclusion

We have formalized a path sensitive type system for the verification of FSM based resource usage properties, in the presence of dynamic resource allocations and aliases. The correctness proof and an inference algorithm are presented.

Acknowledgements. We thank Kwangkeun Yi, Hongseok Yang, Oukseh Lee and the anonymous reviewers for their valuable comments and suggestions. This work was supported by the KOSEF through the AITRC.

References

1. A. Aiken, M. Fähndrich, and R. Levien. Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages. In *PLDI*, 1995.
2. T. Ball and S. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *POPL*, 2002.
3. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. In *ICSE*, 2003.
4. E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, 2004.

5. M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *PLDI*, 2002.
6. R. DeLine and M. Fähndrich. Enforcing High-Level Protocols in Low-Level Software. In *PLDI*, 2001.
7. N. Dor, S. Adams, M. Das, and Z. Yang. Software Validation via Scalable Path-Sensitive Value Flow Analysis. In *ISSTA*, 2004.
8. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *OSDI*, 2000.
9. M. Fähndrich and R. DeLine. Typestates for Objects. In *ECOOP*, 2004.
10. J. Field, D.Goyal, G.Ramalingam, and E.Yahav. Typestate Verification: Abstraction Techniques and Complexity Result. In *SAS*, 2003.
11. C. Flanagan and M. Abadi. Object Types Against Races. In *CONCUR*, 1999.
12. C. Flanagan and M. Abadi. Types for Safe Locking. In *ESOP*, 1999.
13. C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. State. Extended Static Checking for Java. In *PLDI*, 2002.
14. J. Foster, T. Terauhi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *PLDI*, 2002.
15. S. Freund and J. Mitchell. The Type System for Object Initialization in the Java Bytecode Language. *TOPLAS*, 21(6):1196–1250, November 1999.
16. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *In 10th International SPIN Workshop*, 2003.
17. A. Igarashi and N. Kobayashi. Resource Usage Analysis. In *POPL*, 2002.
18. Hyun-Goo Kang, Youil Kim, Taisook Han, and Hwansoo Han. A Path Sensitive Type System for Resource Usage Verification of C like Languages. http://pllab.kaist.ac.kr/∼hgkang/pruv-tm.pdf, 2005.
19. N. Kobayashi. Quasi-Linear Types. In *POPL*, 1999.
20. N. Kobayashi. Time Regions and Effects for Resource Usage Analysis. In *TLDI*, 2003.
21. C. Laneve. A Type System for JVM Threads. *Theoretical Computer Science*, 290(1):741–778, January 2003.
22. O. Lee, H. Yang, and K. Yi. Inserting Safe Memory Reuse Commands into ML-like Programs. In *SAS*, 2003.
23. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, 2001.
24. M. Shapiro and S. Horwitz. The Effects of the Precision of Pointer Analysis. In *SAS*, 1997.
25. G. Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping.* PhD thesis, Cornell University, August 1991.
26. SPEC. SPEC Benchmarks Suite. http://www.spec.org/.
27. R. Strom and S. Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, January 1986.
28. G. Tan, X. Ou, and D. Walker. Enforcing Resource Usage Protocols via Scoped Methods. In *FOOL*, 2003.
29. M. Tofte and L. Birkedal. A Region Inference Algorithm. *TOPLAS*, 20(4):734–767, July 1998.
30. D. Walker, K. Crary, and G. Morriset. Typed Memory Management via Static Capabilities. *TOPLAS*, 22(4):701–771, July 2000.
31. Y. Xie and A. Aiken. Scalable Error Detection using Boolean Satisfiability. In *POPL*, pages 351–363, 2005.

Termination Analysis of Higher-Order Functional Programs

Damien Sereni[1] and Neil D. Jones[2]

[1] Oxford University Computing Laboratory
dsereni@comlab.ox.ac.uk
[2] DIKU, University of Copenhagen
neil@diku.dk

Abstract. Size-change termination (SCT) automatically identifies termination of first-order functional programs. The SCT principle: *a program terminates if every infinite control flow sequence would cause an infinite descent in a well-founded data value* (POPL 2001).

More recent work (RTA 2004) developed a termination analysis of the pure untyped λ-calculus using a similar approach, but an entirely different notion of size was needed to compare higher-order values. Again this is a powerful analysis, even proving termination of certain λ-expressions containing the fixpoint combinator Y. However the language analysed is tiny, not even containing constants.

These techniques are unified and extended significantly, to yield a termination analyser for higher-order, call-by-value programs as in ML's purely functional core or similar functional languages. Our analyser has been proven correct, and implemented for a substantial subset of OCaml.

1 Introduction

Background. Termination proofs are an essential part of program verification and theorem proving. Despite its status as the canonical undecidable problem, automatic termination analysis is a useful tool.

The size-change termination principle of [10] applies to functional programs in which all datatypes are well-founded (which leaves all datatypes but integers and floats, so that natural numbers must be used in lieu of integers). It provides a fully automatic test for termination that does not rely on explicit lexicographic or other orders supplied by the user. The approach was adapted to a termination analysis of the pure untyped λ-calculus [8], using a new well-founded size ordering defined on higher-order values. Analysis of a λ-expression first produced a control flow graph together with some size-change graphs, and then applied the SCT principle.

Contributions of this paper

– A new call-by-value higher-order termination analysis of purely functional languages such as (subsets of) ML or Scheme. The first-order and λ-calculus

analyses of [10, 8] are extended to handle *higher-order values* (not in [10]), and *named functions*, *user-defined datatypes* and *general recursion* (not in [8]).
- A *depth parameter k* extends Shivers' k-CFA [17] to trace flow of data values. This adjustable parameter steers the tradeoff between analysis precision and time complexity. The class of programs recognised as terminating strictly increases as this parameter grows.
- Depth-0 analysis encompasses the analyses of [10, 8]. Depth-1 analysis can prove termination of yet more sophisticated recursions, as well as call-by-value translations of many lazy functional programs.
- The analysis has been implemented for a subset of OCaml, and the implementation is freely available at [15].

1.1 The Core Language

Our termination analysis applies to a purely functional call-by-value language. This paper uses a very restricted *core language* with curried function definitions that is powerful enough to serve as an intermediate representation for most call-by-value functional languages. The abstract syntax of the core language follows in an ML-like datatype declaration:

$$\begin{array}{rll}
\mathit{Program} = & \mathit{FunctionDef\ list} \times \mathit{Expr} & \mathit{def}_1\ \mathit{def}_2 \cdots \mathit{def}_n;\ e \\
\mathit{FunctionDef} = & \mathit{Name} \times \mathit{Name\ list} \times \mathit{Expr} & f\ x_1 \cdots x_n = e \\
e \in \mathit{Expr} = & \ \mathit{Name} & x \\
 & \mid \mathit{FunctionName} & f \\
 & \mid \mathit{Const}\ \mathbf{of}\ \mathit{Constant} & c \\
 & \mid \mathit{App}\ \mathbf{of}\ \mathit{Expr} \times \mathit{Expr} & e_1\ e_2 \\
 & \mid \mathit{If}\ \mathbf{of}\ \mathit{Expr} \times \mathit{Expr} \times \mathit{Expr} & \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2
\end{array}$$

A program is a list of top-level function declarations, together with an expression to evaluate in the context of these definitions. Expressions e are standard and so not explained further. A core language constant may be atomic, e.g., a natural number 0 or 1; or a primitive operator, e.g., $+, -$ or, as in ML, list constructors $[], ::$ and *hd*, *tl*, *null*. Numeric values are assumed well-founded so evaluation of expression $0 - 1$ will cause termination (abortion). We write $\overline{f}$ for the body e of a function f defined by $f\ x_1 \cdots\ x_n = e$.

Translation from a reasonably large OCaml subset to the core language is sketched in Section 5.1, with more details available in the companion report [16]. Thus we use OCaml syntax where convenient in examples.

The standard function *map* on lists illustrates the syntax. (The superscripts are expression labels and can be ignored for now.)

$$\begin{array}{ll}
\mathbf{let\ rec}\ \mathit{map}\ f\ \mathit{xs} = \mathbf{match}\ \mathit{xs}\ \mathbf{with} & [] \rightarrow [] \\
 & \mid x :: \mathit{xs} \rightarrow f\ x ::{}^1 \mathit{map}\ f\ \mathit{xs}
\end{array}$$

This can be translated to the core language program:

$$\mathit{map}\ f\ \mathit{xs} = \mathbf{if}\ \mathit{null}\ \mathit{xs}\ \mathbf{then}\ []\ \mathbf{else}\ f\ (\mathit{hd}\ \mathit{xs}) ::{}^1 \mathit{map}\ f\ (\mathit{tl}\ \mathit{xs})$$

We write f_i to describe a value resulting from applying function f to a length-i argument list. Thus map_0 is the usual map function and map_1, for example, describes the value of map (**fun** $x \to x+1$).

A simple first-order program computes the Ackermann function:

$$\begin{aligned} ack\ m\ n = & \ \textbf{if } m = 0 \textbf{ then } n+1 \textbf{ else} \\ & \ \textbf{if } n = 0 \textbf{ then } {}^1 ack\ (m-1)\ 1 \textbf{ else} \\ & \ {}^2 ack\ (m-1)\ ({}^3 ack\ m\ (n-1)) \end{aligned}$$

1.2 Some Examples of Programs Successfully Analysed

Our system recognises as terminating *all the first-order examples* from [10], including the Ackermann program.

Higher-Order Functions. First, an example from [8] (presented in OCaml-like syntax) computes $f\ n\ x = x + 2^n$ by nontrivial use of recursion and higher-order functions:

$$\begin{aligned} \textbf{let } g\ \ r\ a \ &= \ r\ (r\ a) \\ \textbf{let rec } f\ \ n \ &= \ \textbf{if } n = 0 \textbf{ then } (\textbf{fun } x \to x+1) \textbf{ else } g\ (f\ \ (n-1)) \end{aligned}$$

A number of higher-order functions are in widespread use in functional programs, and as such are of particular interest. In particular, the standard functionals *map*, *foldl* (fold left) and *foldr* (fold right) on lists. We will present a *whole-program analysis*, so calls of form *map f xs* where f is an unknown function parameter cannot occur.

A complication: not every call to *map* need terminate. For example, the function $f\ x = hd\ \ (map\ f\ [x])$ is not terminating on any input. It is straightforward to find a sufficient (but not necessary) condition for an occurrence of *map* to be size-change terminating. The expression *map e xs* is size-change terminating (at depth $k = 0$) for any list value of *xs*, provided

– e is a size-change terminating expression; and
– if e can evaluate to a value described by g_i, then no program-computable value described by g_i+1 can call map_2.

The first condition is clearly necessary. The second condition is stronger than it could be, but serves to exclude functions such that f above that are indirectly recursive through a call to *map*.

Similar results hold for *foldl* and *foldr*. In this sense, the following functions are all size-change terminating at depth $k = 0$ by the higher-order SCT analysis:

$$\begin{aligned} foldr\ op\ \ a\ \ x{::}xs &= \textbf{if } null\ xs \textbf{ then } a \textbf{ else } op\ x\ \ (foldr\ op\ a\ xs) \\ foldl\ op\ \ a\ \ x{::}xs &= \textbf{if } null\ xs \textbf{ then } a \textbf{ else } foldl\ op\ \ (op\ \ a\ \ x)\ \ xs \\ reverse\ xs &= foldl\ (\textbf{fun } ys\ x \to x :: ys)\ [\,]\ xs \\ (@)\ xs\ ys &= foldr\ (::)\ xs\ ys \\ concat\ xss &= foldr\ (@)\ xs\ [\,] \end{aligned}$$

2 Semantic Issues

The number n of arguments to a program defined function or a constant a is called its *arity*, written $\sharp a$. Thus $\sharp \mathit{0} = \sharp[\,] = 0$, and $\sharp \mathit{::} = 2$, and $\sharp f = n$ if there is a function definition of form $f\ x_1 \ldots x_n\ = \ e$. This notation is extended to expressions. We assume that expressions are uniquely labelled, so that the function definition f enclosing an expression e is well-defined. We then define $\sharp e := \sharp f$. Thus $\sharp e$ is the number of bound variables in a complete environment for evaluating e.

The core language is given a standard call-by-value operational semantics, based on closures [9]. A *state* consists of an expression or an atomic constant, together with an *environment* mapping free variables to values. We represent variables positionally for simplicity so an environment is just a list of values, and thus states are written $s = a : v_1 \cdots v_k$ or $s = a : vs$. Notationally we write environments as sequences (strings), using ϵ for the empty environment, juxtaposition for concatenation, and $|vs|$ for the environment's length.

A *value* v is a state that does not require further evaluation because a is an atomic constant, or it has form $s = a : v_1 \cdots v_k$ with $k < \sharp a$. Examples: $\mathit{42} : \epsilon$ or $\mathit{::} : 7$ or $\overline{\mathit{map}} : (\overline{\mathit{successor}} : \epsilon)$. More formally:

$$\begin{aligned}
s \in \mathit{State} &= \{a : vs \mid a \in \mathit{Expr} \cup \mathit{Constant} \wedge vs \in \mathit{Env} \wedge |vs| \leq \sharp a\} \\
vs, ws \in \mathit{Env} &= \mathit{Value}^* \\
v, w \in \mathit{Value} &= \{a : vs \in \mathit{State} \mid |vs| < \sharp a \vee (a \in \mathit{Constant} \wedge \sharp a = 0)\}
\end{aligned}$$

Note that constants and expressions are distinguished in the semantics; this will be useful in the sequel.

The operational semantics defines the *evaluation judgement* $s \Downarrow v$ (s reduces to v, where s is a state and v a value). This is straightforward, with semantic rules for variable bindings and function application shown in Figure 1. Rules for constants, primitive operator evaluation, etc. **if** are usual and may be seen in [16].

$$\frac{\mathsf{Value}}{s \Downarrow s}\, {\scriptstyle s \in \mathit{Value}} \qquad \frac{\mathsf{FunctionRef}}{f : vs \Downarrow \overline{f} : \epsilon} \qquad \frac{\mathsf{Variable}}{x_i : vs \Downarrow vs_i}$$

$$\mathsf{App}\ \frac{e_1 : vs \Downarrow e_b : v_1 \cdots v_n \qquad e_2 : vs \Downarrow v \qquad e_b : v_1 \cdots v_n v \Downarrow w}{e_1 e_2 : vs \Downarrow w}\, {\scriptstyle n < \sharp e_b}$$

Fig. 1. Operational Semantics (excerpt)

Lemma 1. *For any state $a : vs$ reachable in the execution of a program P, a is either a constant or a subexpression of P.*

This result is crucial to our analyses. It holds because of the use of environments to represent bindings, rather than substitution. It provides a natural finite set of *program points*, namely the subexpressions of right-side expressions appearing in the program's function definitions.

Of course the number of constants and, as well, the number of environments can grow unboundedly, so some abstraction will be required for program analysis. This justifies treating constants and expressions differently in the semantics: there are infinitely many potential constants, but only finitely many subexpressions of the program (program points).

Sequentialising Nontermination. The big-step semantics of Figure 1 defines the evaluation rules clearly, but is not particularly helpful for termination analysis. Reason: a nonterminating evaluation of a state s is represented by the *absence* of any proof tree for a judgement $s \Downarrow v$, which provides no immediate handle on the nonterminating computation.

To remedy this, we add small-step semantic rules to trace control flow more exactly in a computation. The extended semantics (following [8], with details in [16]) defines two judgements: $s \Downarrow v$ as above, and $s \to s'$ (s calls s').

The relation $s \to s'$ holds iff in order to conclude $s \Downarrow v$ for some v it is necessary to find v' such that state $s' \Downarrow v'$. This occurs precisely when $s' \Downarrow \_$ occurs as a premise of the inference rule with conclusion $s \Downarrow \_$.

For rule App this adds the call. $e_1e_2 : vs \to e_1 : vs$, Furthermore, if the first premise of the App rule is satisfied (resp. first and second premises), the calls $e_1e_2 : vs \to e_2 : vs$ and $e_1e_2 : vs \to e_b : v_1 \cdots v_n v$ are added (respectively, with notation as in Figure 1).

Definition 1. *The* dynamic call graph *is* $DCG = (State, \to)$ *with states as nodes, and edges* $s \to s'$ *as just defined.*

The transformed semantics now explicitly represents nontermination:

Lemma 2. *For any state s, $s \Downarrow v$ for some value v iff there does not exist an infinite sequence of calls $s = s_0 \to s_1 \to s_2 \to \cdots$.*

This result characterises nontermination as the existence of an infinite path in the dynamic call graph defined by the $\to$ relation. Our termination analysis works by showing that such an infinite path cannot exist.

3 Size-Change Termination

In this section, we lay the groundwork for the higher-order size-change termination analysis. For generality, these are parametrised on some of the analysis choices made, e.g., the precision of the control-flow analysis and the choice of nodes in the size-change graphs. This allows further variations on the analysis to be defined. We shall complete the description in Section 4.

The SCT analysis has been described in [10] for first-order programs, and an analysis of the pure λ-calculus was described in [8]. However, in order to generalise the SCT method, and to enable a more precise analysis, the SCT framework must be generalised substantially from previous treatments.

3.1 The Size-Change Principle

The basis of this analysis is the size-change termination principle [10]: *a program terminates if every infinite control flow sequence would result in infinite descent in at least one well-founded data value.* As remarked previously, this relies on all data value sets being well-founded[1].

In our setting Lemma 2 allows a reformulation: If every infinite *call sequence* causes infinite descent in at least one data value, by well-foundedness no infinite call sequences can occur, so the program terminates.

3.2 Size-Change Graphs

Graph Bases The SCT analysis of [10] operates by annotating each call with some information about size changes. To extend this to the current context we must decide which components of states we are interested in tracking.

As a simple example, consider the Ackermann program of Section 1.2. In this case it is natural to track size changes between the m and n parameters across calls. Function calls in the Ackermann program are of the form $e : v_1 v_2 \rightarrow \overline{ack} : w_1 w_2$ where e is a subexpression of the body of *ack*, and v_i and w_i are natural numbers (i = 1,2). The values v_1 and v_2 represent the values of m and n (respectively) in the calling state, while w_1 and w_2 are the values of m and n in the callee state. As *ack* is a first-order program, the environment is flat (all bound values are simply constants). In the higher-order case, values bound in the environment can be arbitrarily complex.

Definition 2. *The graph basis function gb : State $\rightarrow \mathcal{P}(\{0, 1, \ldots, nv\}^*)$, where nv is the maximum n such that P has a definition $f\ x_1 \ldots\ x_n = e$ and * denotes Kleene closure. For any state s, $gb(s)$ is defined to be the set of* environment paths *of s:*

$$gb(a : v_1 \cdots v_n) = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{i\ p \mid p \in gb(v_i)\}$$

As noted before, for higher-order programs graph bases can contain paths of arbitrary lengths. As an example, consider the state $s = \overline{map} : (\overline{add} : (1 : \epsilon)), ([] : \epsilon)$ (which might occur in the evaluation of *map* (*add* 1) *xs*). Then $gb(s) = \{\epsilon, 1, 1.1, 2\}$. The environment path 1.1 denotes the first parameter of *add* (variables identified positionally).

Each path p in $gb(s)$ denotes a substate of s, namely the value bound in the environment at path p. This is made precise below:

$$\begin{aligned}(a : v_1 \cdots v_n)\nabla\epsilon &= (a : v_1 \cdots v_n)\\ (a : v_1 \cdots v_n)\nabla(i\ p) &= v_i \nabla p\end{aligned}$$

[1] Avery [1] is currently working on extending the size-change approach to the integers.

Size Ordering. The SCT analysis relies on a well-founded order on values. We write this ordering as $<$ throughout. We postpone the definition of this order until the next section, and merely rely on this property:

$$< \text{ is a well-founded order on the set } \mathit{Value} \tag{1}$$

This is a parameter to the general SCT framework, and different SCT-based analyses can use different orders.

Size-Change Graphs. Information about size changes in a call $s \to s'$ takes the form of a *size-change graph.*

Definition 3. *A* size-change graph *between states s and s' (or of type $s \to s'$) is a subset of $gb(s) \times \{\geq, >\} \times gb(s')$.*

The intention is that an edge $(x, >, y)$ in a size-change graph of type $s \to s'$ should indicate that the substate of s at path x is larger than the substate of s' at path y. This is formalised below:

Definition 4. *A size-change graph γ is* safe *for (s,s') if: $(x, >, y) \in \gamma$ implies $s\nabla x > s'\nabla y$, and $(x, \geq, y \in \gamma)$ implies $s\nabla x \geq s'\nabla y$.*

There may be many safe size-change graphs for any call (in particular, any subset of a safe graph is safe); our goal will be to produce graphs that are safe and as precise as possible.

For example, consider (in the Ackermann program of Section 1.2) a call $\textsc{Ack} : m = 3, n = 4 \to \textsc{Ack} : m = 3, n = 3$ (where $\textsc{Ack}$ abbreviates the body of the *ack* function). The maximal safe size-change graph for this call is $\gamma = \{(m, \geq, m), (n, >, n)\}$, using variable names in lieu of their position. More formally, we should write $\gamma = \{(1, \geq, 1), (2, >, 2)\}$. Any subset of γ is safe.

3.3 Recognising Size-Change Termination

The Static Call Graph. The SCT analysis relies on a finite approximation named the *static call graph* to the dynamic call graph of Definition 1. The precise construction of the static call graph is the final parameter of the analysis.

Definition 5. *A* static call graph *is a tuple $SCG = (\mathcal{A}, \to_{scg}, \alpha)$ where $\mathcal{A}$ is a* finite *set of nodes called* abstract states*, $\to_{scg} \subseteq \mathcal{A} \times \mathcal{A}$ is called the* abstract transition relation*, and $\alpha : \mathit{State} \to \mathcal{A}$ is called the* abstraction function.

The first essential property of the static call graph is that it correctly and finitely approximates the transitions of the dynamic call graph.

Definition 6. *The static call graph is* control-safe *for program P if for every $s, s' \in \mathit{State}$ we have*

$$s \to s' \in DCG \textit{ implies } \alpha(s) \to_{scg} \alpha(s') \in SCG$$

We now elaborate this definition to encompass data flow as well as control flow. The following assigns a graph basis to each abstract state, and a size-change graph $G(t \rightarrow_{scg} t')$ to each static transition in *SCG*. The net effect is a finite but two-level graph. The first-level graph $(\mathcal{A}, \rightarrow_{scg}, \alpha)$ has nodes to represent all dynamic states as in Definition 5. Further, with each dynamic control flow transition $s \rightarrow s'$ there is an associated size-change graph $G(\alpha(s) \rightarrow_{scg} \alpha(s'))$ that is required to safely approximate the data flow in $s \rightarrow s'$.

Definition 7. *A tuple* $SCG = (\mathcal{A}, \rightarrow_{scg}, \alpha, gb, G)$ *is an* annotated static call graph *if* $(\mathcal{A}, \rightarrow_{scg}, \alpha)$ *is as in definition 5. The new components:* $gb : State \rightarrow FinSet(\{0, 1, \ldots\}^*)$ *is called the* abstract graph basis*, and* G *associates with each static call graph edge* $t \rightarrow_{scg} t'$ *a size-change graph* $G(t \rightarrow_{scg} t') \subseteq gb(t) \times \{\geq, >\} \times gb(t')$.

Finally, we ensure that the annotated static call graph correctly describes the program's flow of data values.

Definition 8. *The static call graph SCG is* data-flow safe *for program P if it is control-safe for P, and for every* $s, s' \in State$ *we have*

1. $gb(\alpha(s)) \subseteq gb(s)$
2. $s \rightarrow s' \in DCG$ *and* $t = \alpha(s), t' = \alpha(s')$ *implies* $G(t \rightarrow_{scg} t')$ *is safe for* (s, s') *as in Definition 4.*

Condition 1 ensures that any element of the graph basis of an abstracted state $\alpha(s)$ denotes a valid substate of any state s that it represents. Condition 2 ensures that size-change graphs in the annotated static call graphs correctly describe data flow in the dynamic state transitions.

The Finite SCT Criterion. We can now state the central results to apply the SCT principle. More details and proofs may be found in [10, 16]. The criterion for termination that we shall employ is given below; this can be derived from the SCT principle and the definition of safety.

Definition 9. *An infinite sequence of size-change graphs* $(\gamma_i)_{i \in \mathbb{N}}$ *is* infinitely decreasing *if there exists a sequence of nodes* $(x_i)_{i \in \mathbb{N}}$ *and labels* $(l_i)_{i \in \mathbb{N}}$ *such that for each* i, $(x_i, l_i, x_{i+1}) \in \gamma_i$, *and infinitely many of the* l_i *are* $>$ *labels.*

Proposition 1 (SCT). *Let P be a program, and SCG a data-flow safe annotated call graph for P. If for every infinite call sequence* $s_0 \rightarrow_{scg} s_1 \rightarrow_{scg} s_2 \rightarrow_{scg} \ldots$ *the graph sequence* $G(s_0 \rightarrow_{scg} s_1), G(s_1 \rightarrow_{scg} s_2), \ldots$ *is infinitely decreasing, then P terminates.*

The criterion defined in Proposition 1 is stated in terms of all infinite sequences in graph G, and so it is not immediately obvious that this is decidable. However, this is the case, and an algorithm to solve this problem exists [10]:

Theorem 1. *The condition defined in Proposition 1 is decidable (and is PSPACE-complete).*

Despite its high worst-case complexity, this test performs well on natural examples.

4 Higher-Order Programs

In the previous section we have outlined the components of the SCT analysis: a well-founded order on values, a finite set of abstract states (together with their *graph bases*), and a way of constructing the static call graph, annotated with size-change graphs. We shall now describe each of these elements in more detail, completing the overview of this analysis.

4.1 Sizes of Higher-Order Values

The first step is to define the size order $<$ on values. It is straightforward to compare first-order values. For atomic constants (booleans, natural numbers ...) the natural order can be used, as this is well-founded. For constructed constants such as lists, the *structural* order is a reasonable choice: such a value can be written as a term $v = C(w_1, \ldots, w_n)$ for some values w_i. Then certainly $w_i < v$ for each i. The order relation generated by this is suitable [2].

The problem of defining a useful size comparison on higher-order values is of more interest. Recall that such values are represented as closures in our operational semantics, and thus a higher-order value takes the form $e : vs$, where e is an expression, and vs is a list of values. A straightforward but effective order on such values is again structural:

$$v < e : vs \iff v \leq vs_i \text{ for some } i$$

The $<$ relation is the transitive closure of the relation $\{(v, s) \mid v \text{ is bound in the environment of } s\}$ which is a natural basis for comparing values in the higher-order case. The usefulness of this size ordering can be seen from the *foldl'* example (Section 5).

The order on values is then the union of the order on constants and this order on higher-order values. More refined orderings would be possible (for example, an ordering that allows a constant and a functional value to be compared) but this does not appear to increase precision in real cases.

4.2 The Static Call Graph

The key step in the analysis is the construction of the annotated static call graph. Recall that this has a finite set of abstract states as nodes, and call edges annotated by size-change graphs. The data-flow safety criterion (Definition 8) is that the static call graph should overapproximate the DCG, and the size-change graphs should be safe for the transitions they label. We shall describe such a construction, based on a k-CFA [18] of the program.

[2] The fact that this order is well-founded relies on the fact that no cyclic structures can be created in the absence of laziness or reference cells.

The Abstract Domain. The SCG is obtained by abstract interpretation of the operational semantics of the language. We therefore must define the domain of abstract states. The set of exact states is infinite for two separate reasons: the set of constants is infinite, and the *depth* of closures can become unbounded.

Constants can be handled by mapping concrete constants to a finite set of abstract constants. For simplicity we shall assume the following mapping here: all ground constants (such as 42 or []) are mapped to the same value $\bullet$, and all primitive operators (such as $+$ or *::*) are left unchanged.

The abstract domain is then obtained by only retaining information in environments up to an *a priori* fixed depth k. The family of depth-k abstraction functions is thus defined by recursion:

$$\begin{aligned} \alpha_k(c : \epsilon) &= \bullet : \epsilon \quad \text{if } c \in \mathit{Constant} \\ \alpha_0(a : v_1 \cdots v_n) &= a\_n \\ \alpha_{k+1}(a : v_1 \cdots v_n) &= a : \alpha_k(v_1) \cdots \alpha_k(v_n) \end{aligned}$$

Thus $\alpha_k(s)$ is the state obtained from s by turning each constant into its abstract equivalent, and pruning all environment paths of length greater than k. The base case ($k = 0$) discards all information from the environment, but keeps its length to identify partially applied function values.

As an example, consider the state $s = \overline{map} : (\overline{add} : (\mathit{1} : \epsilon)), ([] : \epsilon)$. Then $\alpha_0(s) = map\_2$, $\alpha_1(s) = \overline{map} : (\overline{add}\_1), (\bullet : \epsilon)$, and $\alpha_2(s) = \overline{map} : (\overline{add} : (\bullet : \epsilon)), (\bullet : \epsilon)$.

The graph basis $gb_k(s)$ of a depth-k abstract state is defined as the set of environment paths of length at most $k + 1$:

$$gb_k(\alpha_k(s)) = \{p \in gb(s) \mid |p| \leq k + 1\}$$

This is well-defined, as for any states s and s' with $\alpha_k(s) = \alpha_k(s')$, the sets of environment paths in $gb(s)$ and $gb(s')$ of length at most $k + 1$ are identical. For example, $gb_0(\overline{ack}\_2) = \{\epsilon, 1, 2\}$ (where 1 and 2 identify parameters m and n of *ack* respectively).

Closure Analysis. The construction of a safe call graph by abstract interpretation is now straightforward. We approximate the call and reduction relations ($\rightarrow$ and $\Downarrow$ respectively) in the abstract interpretation, and write $\rightarrow^{scg}$ and $\Downarrow^{scg}$ for their approximations. Abstract values are defined similarly to values in the operational semantic. A depth-0 abstract value is a state of the form e_i, where $i < \sharp e$. A depth-k ($k > 0$) abstract value is of the form $e : vs$ where $|vs| < \sharp e$. Reduction judgements in the abstract interpretation take the form $s \Downarrow^{scg} v$, where s is an abstract state and v an abstract value.

We illustrate the use of *closure analysis* to approximate environments in the depth-0 case. The general case can be found in [16]. The problem that must be solved is to evaluate states of the form x_i_n, where x_i is a function parameter, as no information is kept from the environment. Let f be the function in which the expression x_i occurs (the is well-defined, as expressions are uniquely labelled). Then the only way in which x_i could have been bound to a value v is in the evaluation of a state of the form $e_1e_2\_n$, where $e_1\_n \Downarrow^{scg} \overline{f}\_i - 1$ and $e_2\_n \Downarrow^{scg} v$ (where $n = \sharp e_1e_2$). By the subexpression property (Lemma 1), e_1e_2 is a subexpression of P. This yields the following safe approximation to environment lookups:

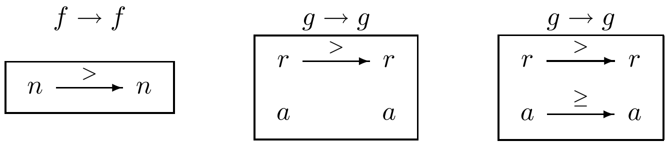
$$\frac{\exists e_1e_2 \in \mathit{subexps}(P) \qquad e_1\_\sharp e_1 \Downarrow^{scg} \overline{f}\_(i-1) \qquad e_2\_\sharp e_2 \Downarrow^{scg} v}{x_i\_\sharp x_i \Downarrow^{scg} v}$$

where as before f is the function the body of which contains the expression x_i. In the general case (depth $k > 0$), the lookup of variable x_i yields a state of depth $k - 1$. This is promoted to a set of states of depth k by expanding the leaves of the tree representing this state to all their possible values, using closure analysis as above.

An Example. We will now illustrate the depth-0 SCG construction for a simple example. Recall the following program (Section 1.2):

$$\begin{array}{rcl} \mathit{succ}\ x &=& x + 1 \\ f\ \ n &=& \textbf{if}\ n = 0\ \textbf{then}\ \mathit{succ}\ \ \textbf{else}\ g\ (f\ (n-1)) \\ g\ \ r\ \ a &=& r\ (r\ a) \end{array}$$

The interesting aspect of this program is function g, which applies its higher-order parameter r. Closure analysis identifies the possible values of $g.r$ as $\overline{succ}\_0$ and $\overline{g}\_1$. There are three self-loops in the SCG, shown below, together with their associated size-change graphs:



$$f \to f \qquad\qquad g \to g \qquad\qquad g \to g$$

$$\begin{tikzcd} n \arrow[r, ">"] & n \end{tikzcd} \qquad \begin{tikzcd} r \arrow[r, ">"] & r \\ a & a \end{tikzcd} \qquad \begin{tikzcd} r \arrow[r, ">"] & r \\ a \arrow[r, "\geq"] & a \end{tikzcd}$$

The decreases in r in both self-calls $g \to g$ occur because in both cases a parameter (r) becomes the top-level state (g_2). A detailed account of how the SCG can be annotated with safe size-change graphs is given in [16]. All three loops in the SCG are therefore size-change terminating, so that this program terminates.

5 Implementation and More Details on Examples

5.1 Implementation

A proof-of-concept implementation of this analysis is freely available [15]. This is based on the OCaml compiler [11] frontend to input OCaml programs, and transforms the programs into the core language. The static call graph is then constructed, together with the size-change graphs annotating edges. Finally, the

SCT criterion [10] is applied to the static call graph, to produce either a guarantee of termination, or a list of loops in the call graph that cannot be proved to terminate.

The input OCaml program is transformed to a core language program by a series of semantic-preserving (and in particular nontermination-preserving) program transformations. The main operations are: eliminating local and anonymous function definitions by λ-lifting [7, 14, 3], and eliminating pattern matching by replacing each **match** statement by a chain of explicit tests.

A substantial concern is whether this analysis is tractable in practice. Ignoring constants for simplicity, the number of possible states for a program of size N, with each function taking at most B arguments is $O((NB)^{B^k})$ for the depth-k analysis. Furthermore, deciding the SCT criterion is a PSPACE-complete problem.

However, in practice the static call graph of real programs is much smaller, so that only a small portion of the potential state space is explored. Also, the SCT criterion is inexpensive in practice. By computing the simply-connected components of the call graph, the SCT test need only be applied to each SCC in turn, and these tend to be small in real programs.

5.2 A Worked Example

To illustrate the SCT analysis, we will now describe its operation on a simple example program (at depth $k = 0$). This program expresses the standard function *foldl* (fold left) on lists in term of its dual fold right. The input program follows:

$$\begin{array}{l}
\textbf{let rec}\ \textit{foldr}\ h\ e = \\
\quad \textbf{function}\ [\,] \rightarrow e \mid x :: xs \rightarrow h\ x\ (\textit{foldr}\ h\ e\ xs) \\
\\
\textbf{let}\ \textit{foldl'}\ op\ e\ xs = \\
\quad \textbf{let}\ id\ x = x\ \textbf{in}\ \ \textbf{let}\ step\ x\ f\ a = f(op\ a\ x)\ \textbf{in} \\
\quad \textit{foldr step id xs e}
\end{array}$$

The definition of *foldr* is standard. The interest in this program lies in the fact that the instance of *foldr* computes a higher-order value, which is applied to e to force evaluation of the accumulated result. This program is size-change terminating, and the operation of the SCT analysis on it is an illuminating example. The first step is to transform this program to the core language, by λ-lifting:

$$\begin{aligned}
\textit{foldr}\ h\ e\ xs &= \textbf{if null}\ xs\ \textbf{then}\ e\ \textbf{else}\ {}^2h\ (\textbf{hd}\ xs)\ {}^3(\textit{foldr}\ h\ e\ (\textbf{tl}\ xs)) \\
id\ x &= x \\
step\ op\ x\ f\ a &= f\ (op\ a\ x) \\
\textit{foldl'}\ op\ e\ xs &= \textit{foldr}\ {}^1(step\ op)\ id\ xs\ e
\end{aligned}$$

What is the control flow of this program? The *step* function applies its parameter f, so the call graph of the program depends on the values that *step.f* may be

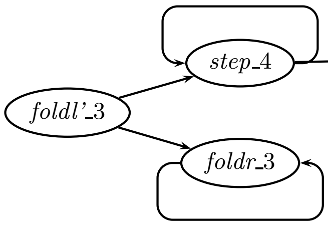


Fig. 2. Simplified Call Graph from *foldl'*

bound to. Closure analysis approximates this. There is one textual application of *step*, in the expression labelled 1. This is passed to the *foldr* function, whence we can deduce that *foldr.h* can take a value described by $\overline{\mathit{step}}\_1$. The h parameter is then applied in expression 2, so that in this application, the expression h (**hd** xs) (which reduces to values of the form $\overline{\mathit{step}}\_2$) is applied to expression 3. This is therefore a binding site for *step.f* identified by closure analysis. The possible values are:

$$\mathit{step.f} \equiv \overline{\mathit{id}}\_0 \quad \text{and} \quad \mathit{step.f} \equiv \overline{\mathit{step}}\_3$$

The call graph of the program can easily be deduced; a simplified representation of it is shown in Figure 2.

There are two self-loops in the call graph. The first is the $\overline{\mathit{foldr}}\_3 \rightarrow \overline{\mathit{foldr}}\_3$ recursion; this is easily seen to terminate, as the list parameter xs of *foldr* decreases at each iteration.

The self-loop $\overline{\mathit{step}}\_4 \rightarrow \overline{\mathit{step}}\_4$ is of more interest. It occurs because of the possible value $\overline{\mathit{step}}\_3$ for f. The size-change graph for this loop is shown below:

$$\begin{tikzcd}
op \arrow[r, "\geq"] & op \\
x & x \\
f \arrow[r, ">"] & f \\
a & a
\end{tikzcd}$$

The *step* loop is thus size-change terminating, as the parameter f decreases. The decrease in f can be explained as follows: the parameter f is applied as an operator in *step*, so all the values bound in the environment of f (two levels down in the environment of *step*) now become bound in the top-level environment of the callee *step* state. In particular, the value of f in the callee state is a strict substate of the value of f in the caller state, hence the decreasing edge.

We have therefore shown that the *foldl'* function is size-change terminating. This illustrates the use of the ordering on functions, as well as closure analysis. Of course, it should be stressed that such informal reasoning is for illustrative purposes only; in actual fact this result was obtained automatically.

5.3 Further Examples

The size-change termination analysis has proved successful on practical examples. In this section we shall show some examples of the SCT analysis in use.

Higher-Order Programs. The standard functions on lists (Section 1) are commonly used, but proving termination of these is not too difficult. We now consider some programs that illustrate the effectiveness of the order on functional values in proving termination.

Ackermann, Higher-Order. This example illustrates the effect of the depth parameter k, and is adapted from an example of the use of the λ-calculus SCT analysis [8], computing the *ack* function using the fixpoint combinator Y (in call-by-value form) explicitly.

$$\begin{aligned}
&\textbf{let } Y\ f = (\textbf{fun } q \to f\ (\textbf{fun } s \to q\ q\ s))\\
&\qquad\qquad (\textbf{fun } q \to f\ (\textbf{fun } s \to q\ q\ s))\\
&\\
&\textbf{let } h\ b\ f\ n = \textbf{if } n = 0 \textbf{ then } f\ 1\\
&\qquad\qquad\quad \textbf{else } f\ (b\ f\ (n-1))\\
&\textbf{let } g\ a\ m = \textbf{if } m = 0 \textbf{ then } (\textbf{fun } v \to v + 1)\\
&\qquad\qquad \textbf{else } Y\ h\ (a\ (m-1))\\
&\\
&\textbf{let } \mathit{ack}\ m\ n = Y\ g\ m\ n
\end{aligned}$$

The depth-0 analysis of this program *fails.* This can be explained by the fact that the two loops introduced by uses of Y in the program are merged, due to the lack of precision. This problem can be solved by using two instances of the Y combinator [8].

However, the depth-1 analysis successfully proves termination, without having to duplicate the Y function. The information kept in the environment allows the two uses of Y to be separated – specifically, the analysis tracks which function was passed to Y as f, if not the value of its arguments.

Lazy Functional Programs: **repmin**. The *repmin*[2] program illustrates the fact that *lazy* functional programs can be analysed within our framework. *repmin* replaces all values stored in the leaves of a binary tree by the least such value, using a circular definition to avoid traversing the tree twice. It is shown below in its lazy version (in Haskell-like syntax to avoid confusion with strict programs), assuming a suitable type of trees:

$$\begin{aligned}
\mathit{repmin}\ t &= \mathit{mt} \textbf{ where } (\mathit{mt}, m) = \mathit{rpm}\ m\ t\\
\mathit{rpm}\ m\ (\mathit{Tip}\ x) &= (\mathit{Tip}\ m, x)\\
\mathit{rpm}\ m\ (\mathit{Fork}\ t_1\ t_2) &= (\mathit{Fork}\ \mathit{mt}_1\ \mathit{mt}_2, m_1 \,\textbf{min}\, m_2)\\
&\qquad \textbf{where } (\mathit{mt}_1, m_1) = \mathit{rpm}\ m\ t_1\\
&\qquad\qquad\quad (\mathit{mt}_2, m_2) = \mathit{rpm}\ m\ t_2
\end{aligned}$$

This program relies crucially on laziness, so that m can be passed as an argument to *rpm* before it is evaluated (as it is returned by *rpm*).

The key observation is that a program terminates under lazy evaluation iff it terminates under call-by-name. We use a well-known encoding of call-by-name in a call-by-value language. A value v that is passed by name is represented by a suspension, that is a function $\lambda().v$ that evaluates v on demand, and is forced when it is used. The function *force* $x = x$ () achieves this. As this encoding is standard, we omit the translated program.

The *repmin* program is size-change terminating at depth $k = 1$. This is similar to the higher-order fold-left case, and again derives from the ordering on functional values. The main difference is that where *foldl'* builds a closure that is isomorphic to the input list, *rpm* builds a closure that is isomorphic to the input tree.

This application of the higher-order SCT analysis is particularly fruitful, as termination of nontrivial lazy functional programs is often hard to establish. In particular, termination of circular programs such as *repmin* depends crucially on evaluation of values never being forced before they are available. The SCT analysis can further be used to prove termination of algorithms operating on lazy data structures (a wealth of such algorithms can be found in [13]).

6 Related Work and Conclusion

We have described a termination analysis for higher-order functional call-by-value languages, such as the purely functional subsets of ML and Scheme. This is very successful on a wide range of higher-order programs. We have also shown that this extends to lazy languages, proving termination of programs for which this is a non-trivial property. We now briefly describe related work, and future directions.

6.1 Related Work

Size-Change Termination. This work is based on the size-change termination principle, originally described for first-order functional programs in [10]. Its extension to the pure λ-calculus [8] introduced the idea of comparing functional values using the structural order on environments. The depth-0 analysis is at least as powerful as the analysis of [8] on λ-terms, though it is applicable to a larger language.

As well as dealing with constants and general recursion, our approach adds the depth parameter as a way of controlling the precision of the analysis. Increasing the depth strictly increases the class of programs that are found to terminate. Depths of 1 and 2 have been found very useful in practice.

Termination Analyses. Much of the work on termination analysis has been directed towards analysing logic programs for termination, as this is often highly nontrivial. Particularly relevant to the SCT method is the Termilog [12] analyser. The algorithm used by Termilog is a close counterpart of the SCT criterion. In particular, size-change graphs for calls from a function are analogous to the weighted rule graphs of [12], with $\geq$ and $>$ as weights. There is also much interest in termination of term rewriting system. The AProVE system [4] identifies termination (and nontermination) of higher-order, untyped term rewriting systems. The system can be parametrised by the well-founded orders searched to prove that there can be no infinite chains in the dependency graph (similar to the call graph in the functional case). This analysis can also make use of a form of the SCT principle, though there does not seem to be an equivalent to our order on higher-order functions in this context.

6.2 Future Work

Expressive Power. A major direction for future work is to investigate the class of programs that are size-change terminating, to determine the boundaries of the method precisely. In particular, relating the higher-order SCT criterion to type systems that guarantee strong normalisation (from the simply-typed λ-calculus, and up to Girard's system F [5]) is natural. It is known that at any fixed depth, there are simply-typed λ-expressions that are not size-change terminating. It is further known that any terminating λ-expression (without constants) will be found terminating for some choice of the depth k (for, a terminating λ-expression has a finite call graph, which the analysis will explore exhaustively for large enough values of the depth parameter). A more precise characterisation of the set of programs accepted by this analysis is lacking, however.

A related issue is the problem of determining the appropriate choice of the depth parameter. In some cases this is straightforward, for example first-order programs do not benefit from increasing k beyond 0. A systematic procedure for choosing the depth parameter for more programs would be a significant improvement.

Applications. There are many applications of termination analysis, the archetypal application being program verification (proving termination and partial correctness separately). Another important application, particularly of functional languages, lies in *theorem proving.* In a theorem prover such as HOL [6], functions can be introduced into the logic, *provided* a formal proof of termination can be produced (this is necessary to guarantee soundness). HOL currently provides facilities for helping the user in this task, such as Konrad Slind's TFL system [19]. However, in many cases termination cannot be proved fully automatically and the burden falls on the user. We hope that our analysis could apply to automatically prove termination in more cases. To integrate our analysis into (say) HOL, it would be necessary to automatically produce a *formal proof* of correctness, rather than just a yes/no result.

References

[1] J. Avery. Size-change termination for programs with non-well-founded data. Graduate project, University of Copenhagen.

[2] R. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[3] O. Danvy. Lambda-lifting in quadratic time. In *FLOPS*, volume 2441 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2002.

[4] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proceedings of the 5th International Workshop of Frontiers of Combining Systems*, volume 3717 of *Lecture Notes in Artificial Intelligence*, pages 216–231, 2005.

[5] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Cambridge tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[6] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[7] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proceedings of a conference on Functional Programming languages and Computer Architecture.* Springer-Verlag, 1985.

[8] N. D. Jones and N. Bohr. Termination analysis of the untyped λ-calculus. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, pages 1–23, 2004.

[9] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6:308–320, 1964.

[10] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the 28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languagaes*, pages 81–92, New York, NY, USA, 2001. ACM Press.

[11] X. Leroy. *The Objective Caml System: Documentation and User's Manual*, 2004.

[12] N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of prolog programs. In *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 64–67, 1997.

[13] C. Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[14] S. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[15] D. Sereni. Higher-order size-change termination analyser. Available at `http://web.comlab.ox.ac.uk/oucl/work/damien.sereni/hoterm.tar.gz`, 2004.

[16] D. Sereni. Termination analysis of higher-order functional programs. Technical Report PRG-RR-04-20, Oxford University Computing Laboratory, 2004. `http://web.comlab.ox.ac.uk/oucl/publications/tr/rr-04-20.html`.

[17] O. Shivers. Control-flow analysis in scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[18] O. Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, May 1991.

[19] K. Slind. *Reasoning about Terminating Functional Programs.* PhD thesis, Institut für Informatik der Technischen Universität München, 1999.

Heterogeneous Fixed Points with Application to Points-To Analysis

Aditya Kanade, Uday Khedker, and Amitabha Sanyal

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay
{aditya, uday, as}@cse.iitb.ac.in

Abstract. Many situations can be modeled as solutions of systems of simultaneous equations. If the functions of these equations monotonically increase in all bound variables, then the existence of extremal fixed point solutions for the equations is guaranteed. Among all solutions, these fixed points uniformly take least or greatest values for all bound variables. Hence, we call them *homogeneous* fixed points. However, there are systems of equations whose functions monotonically increase in some variables and decrease in others. The existence of solutions of such equations cannot be guaranteed using classical fixed point theory. In this paper, we define general conditions to guarantee the existence and computability of fixed point solutions of such equations. In contrast to homogeneous fixed points, these fixed points take least values for some variables and greatest values for others. Hence, we call them *heterogeneous* fixed points. We illustrate heterogeneous fixed point theory through points-to analysis.

1 Introduction

Many situations can be modeled as solutions of systems of simultaneous equations. If the functions of these equations monotonically increase in all bound variables, then the existence of extremal fixed point solutions for the equations is guaranteed through Knaster-Tarski fixed point existence theorem [14,17]. Among all solutions, these fixed points uniformly take least or greatest values for all bound variables. Hence, we call them *homogeneous*.

However, there are systems whose functions monotonically increase in some variables and decrease in others. Emami's (intraprocedural) points-to analysis [4] exhibits this behavior. This analysis computes a variant of may and must aliases in terms of points-to abstraction. The aliases that hold along some but *not* along all paths are captured by *possibly* points-to relation. The aliases that hold along all paths are captured by *definite* points-to relation. While their algorithm performs a fixed point computation, the monotonicity of the functions is not obvious. Consequently the existence and computability of the fixed point solutions cannot be assumed.

The definite and possible points-to relations have both positive as well as negative dependences amongst themselves. Such *heterogeneous dependences* are inherent to points-to analysis. If these mutual dependences are *consistent* in a manner we define later, then the existence of fixed points can be guaranteed. These fixed points called *heterogeneous fixed points* take least values for some variables and greatest values for others. We generalize Knaster-Tarski fixed point existence theorem [14,17] to heterogeneous fixed points.

In section 2, we show that monotonicity of functions in Emami's points-to analysis is not obvious. We then reformulate points-to analysis so that the heterogeneous dependences can be better understood. In section 3, we identify conditions for consistency of heterogeneous dependences so that the existence of fixed points can be assured. In section 4, we define a property called *heterogeneous monotonicity* which captures the consistency conditions. We also define heterogeneous fixed points and show that the former guarantees the existence of latter. Finally, in section 5, we define the solution of our points-to analysis using heterogeneous fixed point theory.

2 Points-To Analyses

In this section, we show that monotonicity of functions in Emami's points-to analysis is not obvious. We then reformulate the analysis to explicate heterogeneous dependences. A brief overview of Emami's points-to analysis is provided in the appendix.

2.1 Monotonicity Issues in Emami's Points-To Analysis

Emami's points-to analysis [4] computes *points-to relation* between pointer expressions. This relation has elements of the following types:

- *Definite Points-To.* A triple (p_1, p_2, D) holds at a program point if the stack location denoted by p_1 contains address of the stack location denoted by p_2 along *every* execution path reaching that point.
- *Possibly Points-To.* A triple (p_1, p_2, P) holds at a program point if the stack location(s) denoted by p_1 contains address(es) of the stack location(s) denoted by p_2 along *some* execution paths reaching that point.

We abstract the algorithm in [4] as data flow equations. In this paper, we restrict ourselves to intraprocedural analysis and a subset of the language in [4].

The points-to relation at IN of a node i is a confluence of points-to relations at OUT of its predecessors $p_1, \ldots, p_k$.

$$\mathsf{input}_i = \begin{cases} \mathsf{Merge}\left(\mathsf{output}_{p_1}, \ldots, \mathsf{output}_{p_k}\right) & \text{if } i \neq \text{entry} \\ \phi & \text{if } i = \text{entry} \end{cases} \tag{1}$$

where "entry" is the unique entry node of the procedure and the Merge operation [3], defined below, is extended to multiple arguments in an obvious way.

$$\begin{aligned}\mathsf{Merge}(S_1, S_2) = \{&(p_1, p_2, D) \mid (p_1, p_2, D) \in S_1 \cap S_2\} \;\cup \\ &\{(p_1, p_2, P) \mid (p_1, p_2, r) \in S_1 \cup S_2 \wedge (p_1, p_2, D) \notin S_1 \cap S_2\}\end{aligned} \tag{2}$$

Note that the definition of Merge excludes the definite information along all paths from being considered as possible points-to information.

Let (x, y, r) hold before an assignment in node i where r is either D or P.

– If x is only likely to be modified, then after the assignment, x may or may not point to y. Hence the definiteness of its pointing to y must be changed to the possibility of its pointing to y. This is captured by the property $\mathsf{changed\_input}_i$ (ref. Appendix A: 24 and 25).
– If x is definitely modified as a side effect of the assignment, then x ceases to point to y. This is captured by the property $\mathsf{kill\_set}_i$ (Appendix A: 26).

An *R-location* represents the variable whose address appears in the rhs. An *L-location* represents the variable which is being assigned this address. Both of the *L-location* and *R-location* depend on the nature of points-to information and can be either *definite* or *possible*.

An assignment generates definite points-to information between its definite L-locations and definite R-locations. All other combinations between its L-locations and R-locations are generated as possibly points-to information. This is captured by the property $\mathsf{gen\_set}_i$ (Appendix A: 27).

Finally, the points-to information at OUT of node i is

$$\mathsf{output}_i = (\mathsf{changed\_input}_i - \mathsf{kill\_set}_i) \cup \mathsf{gen\_set}_i \tag{3}$$

The existence of a fixed point solution requires that all functions in the system of equations should be monotonic in an appropriate lattice. As [4] does not define a partial order over the points-to information domain, we have to assume it. Since the values being computed are sets of points-to triples, we embed them in a lattice with set inclusion as the natural partial order.

Example 1. Let a node i contain the following assignment : $*x = \&y$. Consider the following two cases:

1. Let $\mathsf{input}_i = \{(d, a, D), (x, b, P), (x, c, P)\}$. Then,

$$\mathsf{output}_i = \{(d, a, D), (x, b, P), (x, c, P), (b, y, P), (c, y, P)\}.$$

2. Let $\mathsf{input}'_i = \mathsf{input}_i \cup \{(x, d, P), (b, y, P), (c, y, P)\}$. The resulting output'_i is

$$\mathsf{output}'_i = \{(d, a, P), (x, b, P), (x, c, P), (x, d, P), (b, y, P), (c, y, P), (d, y, P)\}.$$

Clearly, output_i and output'_i are incomparable and

$$\mathsf{input}_i \subseteq \mathsf{input}'_i \not\Rightarrow \mathsf{output}_i \subseteq \mathsf{output}'_i$$

Hence the flow function is non-monotonic w.r.t. set inclusion as partial order. □

As can be seen from the example, an increase in the possible points-to information in input has increased the possible points-to information in output and has decreased the definite points-to information. Similarly, it can be shown that an increase in the definite points-to information in input results in increase of the definite points-to information in output and decrease of the possible points-to information. This is an instance of heterogeneous dependences. While this behavior is inherent in points-to analysis, the existence of and convergence to a fixed point is not guaranteed in general unless monotonicity of functions can be established.

Consider yet another partial order $\leq$ in which the D/P tags in the points-to triples also determine the ordering.

$$\begin{aligned} S_1 \leq S_2 \Longleftrightarrow ((x,y,P) \in S_1 &\implies (x,y,P) \in S_2) \;\wedge \\ ((x,y,D) \in S_1 &\implies (x,y,r) \in S_2, \text{ where } r = D/P) \end{aligned}$$

While the flow function could be monotonic under this partial order, Merge still exhibits non-monotonicity.

$$\begin{aligned} \mathsf{Merge}\,(\{(x,y,D)\},\phi) &= \{(x,y,P)\} \\ \mathsf{Merge}\,(\{(x,y,D)\},\{(x,y,D)\}) &= \{(x,y,D)\} \end{aligned}$$

Though $\phi \leq \{(x,y,D)\}$, $\mathsf{Merge}(\{(x,y,D)\},\phi) \not\leq \mathsf{Merge}(\{(x,y,D)\},\{(x,y,D)\})$.

In summary, the monotonicity of functions in Emami's analysis has not been addressed and is not obvious. Consequently the existence and computability of the fixed point solution cannot be assumed.

2.2 May-Must Points-To Analysis

We now reformulate Emami's analysis to explicate the heterogeneous dependences. As is customary in data flow analysis [12,11], we associate data flow information with IN and OUT of a node. Let $\mathsf{MustIN}_i/\mathsf{MayIN}_i$ be respectively must and may data flow properties at IN of a node i. Let $\mathsf{MustOUT}_i/\mathsf{MayOUT}_i$ be respectively must and may data flow properties at OUT of a node i. Unlike [4], we compute *inclusive* may information implying that MayIN_i and MayOUT_i information also includes points-to information which holds along all paths reaching node i. Our may information corresponds to both definite and possible information whereas our must information corresponds to definite information only.

Since we use separate data flow properties for may and must information, we do not need the third component of points-to triples (D/P). Let U be the universal set containing all type correct points-to pairs $\langle p_1, p_2\rangle$. The lattice of data flow information is $(\wp(U), \subseteq, \cup, \cap, U, \phi)$, where $\wp(U)$ is power set of U and $\subseteq$ is the partial order. Hereafter, we denote this complete lattice by $(\wp(U), \subseteq)$.

The must L-locations and R-locations are represented by $\mathsf{MustL}_i/\mathsf{MustR}_i$ and the may L-locations and R-locations by $\mathsf{MayL}_i/\mathsf{MayR}_i$. Let x be a variable and '&' and '$*$' respectively be dereferencing and referencing operators. The must and may R-locations and L-locations are defined in Table 1.

The points-to analysis is a forward problem as points-to information flows along the control flow of program. The must points-to problem being an all path problem, MustIN of a node is intersection of MustOUT of all its predecessors. In

Table 1. Definitions of L-locations and R-locations

lhs_i	MustL_i	MayL_i
x	$\{x\}$	$\{x\}$
$*x$	$\{y \mid \langle x,y\rangle \in \mathsf{MustIN}_i\}$	$\{y \mid \langle x,y\rangle \in \mathsf{MayIN}_i\}$

rhs_i	MustR_i	MayR_i
$\&x$	$\{x\}$	$\{x\}$
x	$\{y \mid \langle x,y\rangle \in \mathsf{MustIN}_i\}$	$\{y \mid \langle x,y\rangle \in \mathsf{MayIN}_i\}$
$*x$	$\{z \mid \langle x,y\rangle, \langle y,z\rangle \in \mathsf{MustIN}_i\}$	$\{z \mid \langle x,y\rangle, \langle y,z\rangle \in \mathsf{MayIN}_i\}$

the absence of interprocedural information, MustIN of the entry node is initialized to empty set. The data flow equations for MustIN_i and $\mathsf{MustOUT}_i$ are as follows:

$$\mathsf{MustIN}_i = \begin{cases} \bigcap_{p \in pred(i)} \mathsf{MustOUT}_p & \text{if } i \neq \text{entry} \\ \phi & \text{if } i = \text{entry} \end{cases} \tag{4}$$

$$\mathsf{MustOUT}_i = (\mathsf{MustIN}_i - \mathsf{MustKill}_i) \cup \mathsf{MustGen}_i \tag{5}$$

where $pred(i)$ is set of all predecessors of node i. This is a conventional form of data flow analysis which employs IN, OUT, Gen, and Kill properties. Emami's analysis involves an additional property for the "changed" input set.

Since an assignment potentially updates any of its *may* L-locations, all must points-to pairs from the *may* L-locations are killed. The set of such pairs is denoted by $\mathsf{MustKill}_i$. Further, an assignment generates must points-to pairs between all must L-locations and must R-locations. They are contained in the set $\mathsf{MustGen}_i$.

$$\mathsf{MustKill}_i = \{\langle x,y\rangle \mid \underline{x \in \mathsf{MayL}_i} \ \wedge \ \langle x,y\rangle \in \mathsf{MustIN}_i\} \tag{6}$$

$$\mathsf{MustGen}_i = \{\langle x,y\rangle \mid x \in \mathsf{MustL}_i \ \wedge \ y \in \mathsf{MustR}_i\} \tag{7}$$

The may points-to problem is some path problem and hence, MayIN of a node is union of MayOUT of its predecessors. Again, in the absence of interprocedural information, MayIN of the entry node is initialized to empty set. The data flow equations for MayIN_i and MayOUT_i are as follows:

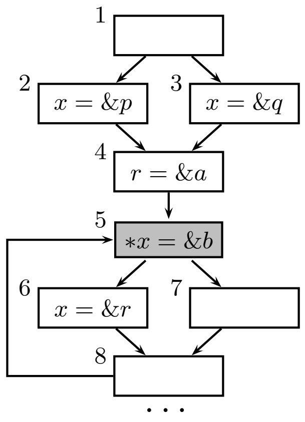
$$\mathsf{MayIN}_i = \begin{cases} \bigcup_{p \in pred(i)} \mathsf{MayOUT}_p & \text{if } i \neq \text{entry} \\ \phi & \text{if } i = \text{entry} \end{cases} \tag{8}$$

$$\mathsf{MayOUT}_i = (\mathsf{MayIN}_i - \mathsf{MayKill}_i) \cup \mathsf{MayGen}_i \tag{9}$$

An assignment kills all may points-to pairs from any *must* L-location of the assignment. The set of such pairs is denoted by $\mathsf{MayKill}_i$. Further, an assignment generates may points-to pairs between all may L-locations and may R-locations. They are contained in the set MayGen_i.

$$\mathsf{MayKill}_i = \{\langle x,y\rangle \mid \underline{x \in \mathsf{MustL}_i} \ \wedge \ \langle x,y\rangle \in \mathsf{MayIN}_i\} \tag{10}$$

$$\mathsf{MayGen}_i = \{\langle x,y\rangle \mid x \in \mathsf{MayL}_i \ \wedge \ y \in \mathsf{MayR}_i\} \tag{11}$$



MustIN_5^1	$\{\langle r,a\rangle\}$
MayIN_5^1	$\{\langle r,a\rangle, \langle x,p\rangle, \langle x,q\rangle\}$
$\mathsf{MustOUT}_5^1$	$\{\langle r,a\rangle\}$
MayOUT_5^1	$\{\langle r,a\rangle, \langle x,p\rangle, \langle x,q\rangle, \langle p,b\rangle, \langle q,b\rangle\}$

After first iteration

MustIN_5^2	$\{\langle r,a\rangle\}$
MayIN_5^2	$\{\langle r,a\rangle, \langle x,p\rangle, \langle x,q\rangle, \langle x,r\rangle, \langle p,b\rangle, \langle q,b\rangle\}$
$\mathsf{MustOUT}_5^2$	ϕ
MayOUT_5^2	$\{\langle r,a\rangle, \langle x,p\rangle, \langle x,q\rangle, \langle x,r\rangle, \langle p,b\rangle, \langle q,b\rangle, \langle r,b\rangle\}$

After second iteration

Fig. 1. How MayIN_i affects MayOUT_i and $\mathsf{MustOUT}_i$

The may-must analysis is *not* a simple combination of the union and intersection data flow analyses. Usually the dependences among the data flow variables are all positive. In this case, there are negative dependences as well.

3 Consistency of Dependences

The nature of the underlying dependences in points-to analysis brought out by may-must formulation is analyzed in this section. We identify the conditions which guarantee existence of fixed points in presence of such dependences.

3.1 Positive and Negative Dependences

From the underlined terms in the data flow equations (4) – (11), it is clear that $\mathsf{MustOUT}_i$ decreases with increase in MayIN_i and MayOUT_i decreases with increase in MustIN_i.

Definition 1. *A variable x* depends *on a variable y iff x is defined in terms of y and there exist at least two distinct values of y such that the corresponding values of x are distinct, keeping rest of the variables constant.*

If the non-decreasing values of y result in the non-decreasing values of x then x depends *positively* on y. Otherwise, x depends *negatively* on y.

Example 2. Consider the program flow graph shown in Figure 1. To create maximum optimization opportunities, we want the largest set of must points-to pairs and the smallest set of may points-to pairs which together form the solution of may-must analysis. Hence, we initialize the data flow variables as follows:

$$\mathsf{MustIN}_i = \mathsf{MustOUT}_i = U \tag{12}$$

$$\mathsf{MayIN}_i = \mathsf{MayOUT}_i = \phi \tag{13}$$

where U is the universal set containing all type correct points-to pairs.

We compute the data flow properties using round robin iterative method in which nodes are visited in the reverse depth first order. Let P_i^j be a property P at node i in iteration j. The data flow values at node 5 after first and second iterations over the program flow graph are shown in Figure 1.

MustIN_5 remains the same in first and second iterations, but MayIN_5 increases in second iteration which causes $\mathsf{MustOUT}_5$ to decrease. Thus, the dependence of $\mathsf{MustOUT}_i$ on MayIN_i is negative. MayOUT_5 increases in second iteration and hence the dependence of MayOUT_i on MayIN_i is positive. □

Similarly, it can be shown that $\mathsf{MustOUT}_i$ depends positively on MustIN_i and MayOUT_i depends negatively on MustIN_i. The dependences in may-must data flow equations can be summarized as follows:

D1. The dependence of MustIN_i on $\mathsf{MustOUT}_p$, $p \in \mathit{pred}(i)$, is positive (4).
D2. The dependence of $\mathsf{MustOUT}_i$ on MustIN_i is positive but that on MayIN_i is negative (5, 6, and 7).
D3. The dependence of MayIN_i on MayOUT_p, $p \in \mathit{pred}(i)$, is positive (8).
D4. The dependence of MayOUT_i on MayIN_i is positive but that on MustIN_i is negative (9, 10, and 11).

Since not all dependences between the variables are positive, the existence and computability of fixed point solutions of the equations cannot be guaranteed using the classical results [14,13,17]. However, as we demonstrate later if the dependences are mutually consistent, the existence of fixed points can be guaranteed.

3.2 Consistency of Dependences

Consistency of dependences can be defined in terms of a dependence graph. Nodes in this graph represent the bound variables. If a variable x depends positively on a variable y, then there is a solid edge from x to y. If x depends negatively on y, then there is a dashed edge from x to y. If a variable x does not depend on y, then there is no edge from x to y.

The *parity* of a path in a dependence graph is even if the path has an even number of dashed edges, otherwise its parity is odd. If all paths between every pair of nodes have the same parity, then the dependences between those variables are consistent. If the parity is even then an increase in one's value leads to an increase in other's and vice versa. If the parity is odd then an increase in one's value leads to a decrease in other's and vice versa. If the paths are of different parities then the mutual influences cannot be determined.

Definition 2. *Dependences in a system of simultaneous equations are consistent iff for every pair of nodes (x, y) contained in a strongly connected component of the dependence graph, all paths between x and y have the same parity.*

For simplicity, we assume that the dependence graph of the variables has a single maximal strongly connected component. The systems that have more than one maximal strongly connected components in their dependence graphs are discussed in [10].

4 Heterogeneous Fixed Points

In the classical setting, monotonicity and fixed points of a set of functions are straightforward generalizations of the corresponding definitions of the individual functions. These generalizations are *uniform.* Hence, we call them *homogeneous.*

To capture systems like may-must data flow equations, we generalize these formulations so that they *need not* be uniform over the components. We call our formulations *heterogeneous.* Here, we present only relevant part of the formulation and associated results. A more detailed treatment can be found in [10]. We now introduce some terminology used.

Let S_n be a system of n $(n > 0)$ simultaneous equations in n variables:

$$\begin{aligned} x_1 &= f_1(x_1, \ldots, x_n) \\ &\vdots \\ x_n &= f_n(x_1, \ldots, x_n) \end{aligned}$$

The functions $f_1, \ldots, f_n$ are called the *component functions* of S_n. Let $\boldsymbol{F}$ be a function defined as $\boldsymbol{F}(\boldsymbol{X}) = \langle f_1(\boldsymbol{X}), \ldots, f_n(\boldsymbol{X})\rangle$ where $\boldsymbol{X} = \langle x_1, \ldots, x_n\rangle$. We call $\boldsymbol{F}$ the *function vector* of S_n. The variables $x_1, \ldots, x_n$ which appear on the left side of the equalities are called the *bound* variables of S_n.

We assume that a bound variable x_i takes values from a finite[1] complete lattice $L_i = (L_i, \sqsubseteq_i, \sqcup_i, \sqcap_i, \top_i, \bot_i)$, where $\sqsubseteq_i$ is the partial order over the set L_i, $\sqcup_i$ and $\sqcap_i$ are respectively join and meet of the lattice, and $\top_i$ and $\bot_i$ are respectively the top and bottom of the lattice. Let $\boldsymbol{L} = (\boldsymbol{L}, \sqsubseteq, \sqcup, \sqcap, \top, \bot) = L_1 \times \cdots \times L_n$. A function f_i has type $\boldsymbol{L} \to L_i$. The function vector $\boldsymbol{F}$ has type $\boldsymbol{L} \to \boldsymbol{L}$.

4.1 Heterogeneous Monotonicity

Consider a partition (P, Q) of the set $\{1, \ldots, n\}$. We define the heterogeneous monotonicity of a function vector with respect to a partition (P, Q) as follows :

Definition 3. *A function vector* $\boldsymbol{F} : \boldsymbol{L} \to \boldsymbol{L}$ *is* heterogeneously monotonic *(or simply* h-monotonic*) w.r.t. a partition* (P, Q) *iff*

1. *for an* $i \in P$, f_i *monotonically increases in* x_j, *if* $j \in P$ *and monotonically decreases in* x_j, *if* $j \in Q$ *and*
2. *for an* $i \in Q$, f_i *monotonically increases in* x_j, *if* $j \in Q$ *and monotonically decreases in* x_j, *if* $j \in P$.

If the function vector $\boldsymbol{F}$ is h-monotonic w.r.t. a partition (P, Q), then (P, Q) is called a *valid* partition for the system. The classical monotonicity of a function vector $\boldsymbol{F}$ is a special case of h-monotonicity with $(\{1, \ldots, n\}, \phi)$ and $(\phi, \{1, \ldots, n\})$ as (the only) valid partitions.

[1] For simplicity, we consider finite lattices. More general treatment is available in [10].

A function f_i monotonically increases in a variable x_j iff the variable x_i depends positively on the variable x_j. A function f_i monotonically decreases in a variable x_j iff the variable x_i depends negatively on the variable x_j.

Lemma 1. *There exists a valid partition for a system S_n iff the dependences in S_n are consistent.*

Proof. Let there be a valid partition but the dependences in S_n not be consistent. There exist two variables x_i and x_j such that ρ_1 and ρ_2 are two paths between them and the parity of ρ_1 is even and that of ρ_2 is odd. From the definition of h-monotonicity and composibility of the dependences, due to the dependences along the path ρ_1, i and j should belong to the same set of a partition. Similarly, due to the dependences along the path ρ_2, i and j should belong to the different sets of a partition. This is a contradiction.

Let there be no valid partition but the dependences in S_n be consistent. There exist two variables which can be placed in the same as well as the different sets. Clearly, there exist two paths between them which have different parities. Hence, the dependences in S_n are not consistent. $\square$

If i and j belong to the same set in a valid partition, then x_i and x_j have even parity paths between them in the dependence graph. If i and j belong to different sets in a valid partition, then x_i and x_j have odd parity paths between them in the dependence graph. That is, an increase in the value of a variable in a set can lead only to an increase in the values of the variables in that set and decrease in the values of the variables in the other set.

4.2 Identifying Valid Partitions

We give an algorithm called Even-Odd-Analysis to identify the valid partitions given the dependence graph of a system. This algorithm returns valid partitions iff the dependences in S_n are consistent. Let $D = (V, E)$ be the dependence graph of a system of equations, where V is the set of bound variables and E is the set of edges representing the dependences among the variables.

Let *dependence* be a property of edges. A value 1 of *dependence*$(\langle x_u, x_v\rangle)$ denotes a solid edge from x_u to x_v while a value -1 denotes a dashed edge from x_u to x_v. Let *membership*(x_u) denote to which set of the partition the variable x_u belongs. The function Initialize initializes the *dependence* properties according to the monotonicities of the functions and assigns a value 0 to *membership* property of all variables to indicate that their membership in the sets of a valid partition is yet to be determined.

Initialize(D).

```
1.  for each $\langle x_u, x_v\rangle \in E$
2.      if $f_u$ monotonically increases in $x_v$ then
3.          $dependence(\langle x_u, x_v\rangle) \leftarrow 1$ /* solid edge */
4.      else
5.          $dependence(\langle x_u, x_v\rangle) \leftarrow -1$ /* dashed edge */
```

```
 6.  for each $x_u \in V$
 7.      $membership(x_u) \leftarrow 0$
 8.  return
```

Let Strongly-Connected-Components be a function which takes a graph and returns the strongly connected components in it as sets of sets of nodes. Select-Node selects an element from a set. Even-Odd-Analysis first initializes the properties explained above. It then selects a node from a strongly connected component and assigns it *membership* in a set. It invokes a function Dft which traverses the graph in depth-first order and determines memberships of nodes iff the dependences are consistent. If Dft returns a 0 then the dependences are inconsistent and there is no valid partition. Otherwise, the sets of the valid partition can be constructed from the *membership* properties of nodes.

Even-Odd-Analysis(D).

```
 1.  Call Initialize($D$)
 2.  $SCC \leftarrow$ Strongly-Connected-Components($D$)
 3.  for each $C \in SCC$
 4.      $x_u \leftarrow$ Select-Node($C$)
 5.      $membership(x_u) \leftarrow 1$
 6.      $success \leftarrow$ Dft($x_u, C$)
 7.      if $success = 0$ then
 8.          print "No partitions possible for the component [$C$]"
 9.      else
10.          $P \leftarrow \{ u \in \{1, \ldots, n\} \mid x_u \in C \wedge membership(x_u) = 1 \}$
11.          $Q \leftarrow \{ u \in \{1, \ldots, n\} \mid x_u \in C \wedge membership(x_u) = -1 \}$
12.          print "Partitions for component [$C$] are ([$P$], [$Q$]) and ([$Q$], [$P$])"
13.  return
```

Dft takes a node x_u and a strongly connected graph C. For every neighbour x_v of x_u in C, it checks whether x_v has been visited previously. If not then it assigns x_v a membership consistent with the dependence $\langle x_u, x_v \rangle$. If the dependence is 1, then x_v goes to the same set else it goes to the other set. It then calls itself on x_v and C. If failure is returned then it propagates it upwards. Otherwise it analyzes other neighbours of x_u. If x_v has been visited, then it verifies the consistency of membership of x_v w.r.t. the dependence $\langle x_u, x_v \rangle$. If the dependences are inconsistent, it returns a failure, otherwise it goes to next neighbour of x_u. Finally, returns a success status.

Dft(x_u, C).

```
 1.  for each $x_v \in C$ such that $\langle x_u, x_v \rangle \in E$ /* for every neighbour of $x_u$ */
     /* if unvisited, assign membership consistent with dependence $\langle x_u, x_v \rangle$*/
 2.      if $membership(x_v) = 0$ then
 3.          $membership(x_v) \leftarrow dependence(\langle x_u, x_v \rangle) \times membership(x_u)$
 4.          $success \leftarrow$ Dft($x_v, C$) /* traverse recursively */
```

```
5.         if $success = 0$ then return 0 /* propagate failure upwards */
     /* if visited, check for inconsistency with the dependence $\langle x_u, x_v\rangle$ */
6.       elseif $membership(x_v) \neq dependence(\langle x_u, x_v\rangle) \times membership(x_v)$ then
7.             return 0 /* if inconsistent, return the failure status */
8.   return 1 /* return the success status */
```

4.3 Relating to Homogeneity

We now relate heterogeneity with classical homogeneity. We construct a lattice

$$\boldsymbol{L}^{(\mathrm{P,Q})} = \left(\boldsymbol{L}^{(\mathrm{P,Q})}, \sqsubseteq^{(\mathrm{P,Q})}, \sqcup^{(\mathrm{P,Q})}, \sqcap^{(\mathrm{P,Q})}, \top^{(\mathrm{P,Q})}, \bot^{(\mathrm{P,Q})}\right)$$

as a product of the component lattices or their *duals*[2] as defined below:

$$\boldsymbol{L}^{(\mathrm{P,Q})} = \boldsymbol{L}_1^{(\mathrm{P,Q})} \times \cdots \times \boldsymbol{L}_n^{(\mathrm{P,Q})}$$
$$\text{where, } \boldsymbol{L}_i^{(\mathrm{P,Q})} = \begin{cases} L_i & i \in P \\ L_i^{-1} & i \in Q \end{cases}$$

where L_i^{-1} is the dual of L_i and (P, Q) is a partition of $\{1, \ldots, n\}$.

Consider a system $S_n^{(\mathrm{P,Q})}$ whose function vector $\boldsymbol{F}^{(\mathrm{P,Q})} : \boldsymbol{L}^{(\mathrm{P,Q})} \to \boldsymbol{L}^{(\mathrm{P,Q})}$ is isomorphic to $\boldsymbol{F}$. The component functions of $S_n^{(\mathrm{P,Q})}$ are $f_1^{(\mathrm{P,Q})}, \ldots, f_n^{(\mathrm{P,Q})}$. The systems S_n and $S_n^{(\mathrm{P,Q})}$ are called *duals* of each other w.r.t. the partition (P, Q).

Lemma 2. *If the systems S_n and $S_n^{(\mathrm{P,Q})}$ are duals of each other w.r.t. a partition (P, Q), then (P, Q) is a valid partition for S_n iff the function vector $\boldsymbol{F}^{(\mathrm{P,Q})} : \boldsymbol{L}^{(\mathrm{P,Q})} \to \boldsymbol{L}^{(\mathrm{P,Q})}$ of $S_n^{(\mathrm{P,Q})}$ is monotonic.*

Proof. We prove forward implication by considering following two cases:

Case 1. Let $i \in P$. By construction, $\boldsymbol{L}_i^{(\mathrm{P,Q})} = L_i$. Since, the function vector $\boldsymbol{F}$ of S_n is h-monotonic w.r.t. the partition (P, Q),

$$\text{if } j \in P,\ a_j \sqsubseteq_j a'_j \implies f_i(x_1, \ldots, a_j, \ldots, x_n) \sqsubseteq_i f_i(x_1, \ldots, a'_j, \ldots, x_n),$$
$$\text{if } j \in Q,\ a_j \sqsupseteq_j a'_j \implies f_i(x_1, \ldots, a_j, \ldots, x_n) \sqsubseteq_i f_i(x_1, \ldots, a'_j, \ldots, x_n),$$

The component functions f_i and $f_i^{(\mathrm{P,Q})}$ are isomorphic. By construction of the lattice $\boldsymbol{L}^{(\mathrm{P,Q})}$,

$$\text{if } j \in P,\ a_j \sqsubseteq_j^{(\mathrm{P,Q})} a'_j \implies f_i^{(\mathrm{P,Q})}(x_1, \ldots, a_j, \ldots, x_n) \sqsubseteq_i^{(\mathrm{P,Q})} f_i^{(\mathrm{P,Q})}(x_1, \ldots, a'_j, \ldots, x_n)$$
$$\text{if } j \in Q,\ a_j \sqsubseteq_j^{(\mathrm{P,Q})} a'_j \implies f_i^{(\mathrm{P,Q})}(x_1, \ldots, a_j, \ldots, x_n) \sqsubseteq_i^{(\mathrm{P,Q})} f_i^{(\mathrm{P,Q})}(x_1, \ldots, a'_j, \ldots, x_n)$$

Hence, for an $i \in P$, $f_i^{(\mathrm{P,Q})}$ monotonically increases in all bound variables.

Case 2. For an $i \in Q$, $\boldsymbol{L}_i^{(\mathrm{P,Q})} = L_i^{-1}$. By arguments similar to the above case, $f_i^{(\mathrm{P,Q})}$ can be shown to be monotonically increasing in all bound variables.

Thus, all component functions of $S_n^{(\mathrm{P,Q})}$ monotonically increase in all variables and hence, $\boldsymbol{F}^{(\mathrm{P,Q})}$ is monotonic. The converse is by an analogous argument. □

[2] Lattices $(L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ and $(L, \sqsupseteq, \sqcap, \sqcup, \bot, \top)$ are called duals of each other.

Lemma 3. *A system S_n and its dual system $S_n^{(P,Q)}$ w.r.t. any partition (P, Q) of the set $\{1, \ldots, n\}$ have the same solutions.*

Proof. The systems are isomorphic and are defined over the same sets. □

4.4 Heterogeneous Fixed Points

We have assumed that a component lattice L_i is a complete lattice, hence any subset of L_i has a least upper bound (lub) and a greatest lower bound (glb). Let $Fix(\boldsymbol{F})$ be the set of fixed points of $\boldsymbol{F}$. Let $Fix_i(\boldsymbol{F})$ be the set of elements from $\boldsymbol{L}_i$ that belong to some fixed point of $\boldsymbol{F}$.

We define $\text{hfp}^{(P,Q)}(\boldsymbol{F})$ element-wise such that its ith element $\text{hfp}_i^{(P,Q)}(\boldsymbol{F})$ is glb of $Fix_i(\boldsymbol{F})$, if $i \in P$ and lub of $Fix_i(\boldsymbol{F})$, if $i \in Q$.

$$\text{hfp}_i^{(P,Q)}(\boldsymbol{F}) = \begin{cases} \sqcap_i Fix_i(\boldsymbol{F}) \text{ if } i \in P \\ \sqcup_i Fix_i(\boldsymbol{F}) \text{ if } i \in Q \end{cases} \tag{14}$$

We now show that if (P, Q) is a valid partition for a system S_n, then $\text{hfp}^{(P,Q)}$ $(\boldsymbol{F})$ exists and is a fixed point of the function vector $\boldsymbol{F}$. We call this fixed point, a *heterogeneous fixed point* (HFP) of an $\boldsymbol{F}$ w.r.t. a partition (P, Q). From among the fixed point values, it takes element-wise least possible values from the component lattices with subscripts in P and element-wise greatest possible values from the component lattices with subscripts in Q. We now give an existence theorem for heterogeneous fixed points.

Theorem 1 (HFP Existence Theorem). *If the function vector $\boldsymbol{F} : \boldsymbol{L} \to \boldsymbol{L}$ of a system S_n is h-monotonic w.r.t. a partition (P, Q), then*

$$\text{hfp}^{(P,Q)}(\boldsymbol{F}) \in Fix(\boldsymbol{F})$$

Proof. Since the function vector $\boldsymbol{F}$ is h-monotonic w.r.t. a partition (P, Q), from Lemma 2, $\boldsymbol{F}^{(P,Q)} : \boldsymbol{L}^{(P,Q)} \to \boldsymbol{L}^{(P,Q)}$ is monotonic. By Knaster-Tarski fixed point theorem [14,17], the least fixed point of $\boldsymbol{F}^{(P,Q)}$, lfp $\left(\boldsymbol{F}^{(P,Q)}\right)$ exists. We can write lfp $\left(\boldsymbol{F}^{(P,Q)}\right)$ element-wise as:

$$\text{lfp}_i\left(\boldsymbol{F}^{(P,Q)}\right) = \begin{cases} \sqcap_i^{(P,Q)} Fix_i\left(\boldsymbol{F}^{(P,Q)}\right) \text{ if } i \in P \\ \sqcap_i^{(P,Q)} Fix_i\left(\boldsymbol{F}^{(P,Q)}\right) \text{ if } i \in Q \end{cases} \tag{15}$$

From Lemma 3, $Fix(\boldsymbol{F}) = Fix\left(\boldsymbol{F}^{(P,Q)}\right)$. (15) can be rewritten as:

$$\text{lfp}_i\left(\boldsymbol{F}^{(P,Q)}\right) = \begin{cases} \sqcap_i^{(P,Q)} Fix_i(\boldsymbol{F}) \text{ if } i \in P \\ \sqcap_i^{(P,Q)} Fix_i(\boldsymbol{F}) \text{ if } i \in Q \end{cases} \tag{16}$$

From the construction of the dual lattice $\boldsymbol{L}^{(P,Q)}$, $\sqcap_i^{(P,Q)} = \sqcap_i$, if $i \in P$ and $\sqcap_i^{(P,Q)} = \sqcup_i$, if $i \in Q$. From this and (16) and the definition of hfp (14),

$$\text{lfp}_i\left(\boldsymbol{F}^{(P,Q)}\right) = \left\{\begin{matrix} \sqcap_i Fix_i(\boldsymbol{F}) \text{ if } i \in P \\ \sqcup_i Fix_i(\boldsymbol{F}) \text{ if } i \in Q \end{matrix}\right\} = \text{hfp}_i^{(P,Q)}(\boldsymbol{F})$$

Hence, $\text{hfp}^{(P,Q)}(\boldsymbol{F}) \in Fix(\boldsymbol{F})$. □

For simplicity of exposition, we have proved Theorem 1 by appealing to Knaster-Tarski theorem, but it is also possible to prove it from the first principles.

From the construction of $\boldsymbol{L}^{(\mathrm{P},\mathrm{Q})}$, we already know that

$$\begin{aligned} \perp_i^{(\mathrm{P},\mathrm{Q})} &= \perp_i \ \text{ if } i \in P \\ \perp_i^{(\mathrm{P},\mathrm{Q})} &= \top_i \ \text{ if } i \in Q \end{aligned} \tag{17}$$

The least fixed point $\mathrm{lfp}\left(\boldsymbol{F}^{(\mathrm{P},\mathrm{Q})}\right)$ can be computed iteratively starting with $\perp^{(\mathrm{P},\mathrm{Q})}$ [13]. Hence, the heterogeneous fixed point of a function vector $\boldsymbol{F}$ w.r.t. a valid partition (P, Q) can be defined as follows:

$$\mathrm{hfp}^{(\mathrm{P},\mathrm{Q})}(\boldsymbol{F}) = \boldsymbol{F}^k \left(\perp^{(\mathrm{P},\mathrm{Q})}\right) \tag{18}$$

where $k \in \mathbb{N}$ is the least number such that $\boldsymbol{F}^k\left(\perp^{(\mathrm{P},\mathrm{Q})}\right) = \boldsymbol{F}^{k-1}\left(\perp^{(\mathrm{P},\mathrm{Q})}\right)$.

5 Solution of May-Must Data Flow Analysis

We now apply the heterogeneous fixed point theory to may-must points-to analysis (ref. section 2.2). Consider the data flow equations (4), (5), (8), and (9). If there are n nodes in a program flow graph, there are $4 \times n$ equations forming a system $S_{(4\times n)}$. MustIN_i, $\mathsf{MustOUT}_i$, MayIN_i, and MayOUT_i, where $i \in \{1, \ldots, n\}$ are the bound variables of the system. The corresponding flow functions are denoted by $\mathsf{f}_{\mathsf{MustIN}i}$, $\mathsf{f}_{\mathsf{MustOUT}i}$, $\mathsf{f}_{\mathsf{MayIN}_i}$, and $\mathsf{f}_{\mathsf{MayOUT}_i}$.

For convenience, we abstract the data flow equations as:

$$\begin{aligned} x_{(4\times i-3)} &= f_{(4\times i-3)}\left(x_1, x_2, \ldots, x_{4\times n}\right) \\ x_{(4\times i-2)} &= f_{(4\times i-2)}\left(x_1, x_2, \ldots, x_{4\times n}\right) \\ x_{(4\times i-1)} &= f_{(4\times i-1)}\left(x_1, x_2, \ldots, x_{4\times n}\right) \\ x_{(4\times i-0)} &= f_{(4\times i-0)}\left(x_1, x_2, \ldots, x_{4\times n}\right) \end{aligned}$$

For translating these equations back to may-must points-to analysis, we will use the following mappings:

Bound Variables	Functions
$\mathsf{MustIN}_i \leftrightarrow x_{(4\times i-3)}$	$\mathsf{f}_{\mathsf{MustIN}i} \leftrightarrow f_{(4\times i-3)}$
$\mathsf{MustOUT}_i \leftrightarrow x_{(4\times i-2)}$	$\mathsf{f}_{\mathsf{MustOUT}i} \leftrightarrow f_{(4\times i-2)}$
$\mathsf{MayIN}_i \leftrightarrow x_{(4\times i-1)}$	$\mathsf{f}_{\mathsf{MayIN}_i} \leftrightarrow f_{(4\times i-1)}$
$\mathsf{MayOUT}_i \leftrightarrow x_{(4\times i-0)}$	$\mathsf{f}_{\mathsf{MayOUT}_i} \leftrightarrow f_{(4\times i-0)}$

A component lattice is $L_j = (\wp(U), \subseteq)$. The product lattice of the system is $\boldsymbol{L} = L_1 \times \cdots \times L_{(4\times n)} = (\wp(U), \subseteq)^{(4\times n)}$. A component function is of the type $(\wp(U), \subseteq)^{(4\times n)} \rightarrow (\wp(U), \subseteq)$.

Consider the following two sets $P, Q \subseteq \{1, \ldots, (4 \times n)\}$:

$$P = \{(4 \times i - 1),\, (4 \times i - 0) \mid i \in \{1, \ldots, n\}\} \tag{19}$$

$$Q = \{(4 \times i - 3),\, (4 \times i - 2) \mid i \in \{1, \ldots, n\}\} \tag{20}$$

The set P represents variables $\mathsf{MayIN}_i/\mathsf{MayOUT}_i$ and the corresponding functions $\mathsf{f}_{\mathsf{MayIN}_i}/\mathsf{f}_{\mathsf{MayOUT}_i}$. Q represents variables $\mathsf{MustIN}_i/\mathsf{MustOUT}_i$ and the functions $\mathsf{f}_{\mathsf{MustIN}i}/\mathsf{f}_{\mathsf{MustOUT}i}$.

Claim. The function vector $\boldsymbol{F}$ of the system $S_{(4\times n)}$ is h-monotonic w.r.t. the partition (P, Q).

Proof. When a variable x_i does not depend on a variable x_j, then it is safe to assume that f_i either monotonically increases in x_j or monotonically decreases in x_j, i.e. x_i has either a positive or a negative dependence on x_j. The result follows directly from the dependences D1–D4 and the construction of $S_{(4\times n)}$. □

There could be several other valid partitions, but we are interested in this particular partition as we want the largest possible must information and the smallest possible may information, for enabling maximum optimization opportunities. In [10], we give results about the number and the nature of valid partitions for any given system. The desired solution of may-must data flow equations is $\text{hfp}^{(\text{P},\text{Q})}(\boldsymbol{F})$. To compute the solution, we first initialize the variables with corresponding elements in $\bot^{(\text{P},\text{Q})}$ as follows:

$$\begin{aligned} x_{(4\times i-3)} &= \mathsf{MustIN}_i &&= U \\ x_{(4\times i-2)} &= \mathsf{MustOUT}_i &&= U \\ x_{(4\times i-1)} &= \mathsf{MayIN}_i &&= \phi \\ x_{(4\times i-0)} &= \mathsf{MayOUT}_i &&= \phi \end{aligned} \tag{21}$$

With the above initialization, the heterogeneous fixed point solution can be computed by iteratively solving the may-must data flow equations until two consecutive iterations result in same values (ref. (18)).

Further, it can be shown that our analysis and Emami's analysis compute equivalent information. Since our analysis converges, it can be shown that starting with appropriate initializations Emami's analysis also converges [9].

6 Conclusions and Future Work

Many analyses can be modeled as fixed point solutions of systems of simultaneous equations in which the functions may monotonically increase in some bound variables and decrease in others. The classical fixed point theory does not cover such situations. It requires all dependences among the bound variables to be positive. The classical extremal fixed points uniformly take either least values for all variables or greatest values for all variables, where the element-wise comparison is restricted to the set of solutions.

The heterogeneous fixed point theory is a generalization of the classical fixed point theory. It allows positive as well as negative dependences among the variables. We have shown that if the dependences are mutually consistent then the variables can be partitioned into two sets such that two variables belong to the same set iff the dependences between them are positive. This guarantees the existence of a fixed point called heterogeneous fixed point, which depending on the partition, takes least values for some variables from among all fixed points, and greatest values for others. Our theory also suggests appropriate initialization thereby assuring computability of fixed points. We have applied heterogeneous fixed point theory to explain convergence issues in points-to analysis.

Further work includes exploring applications of heterogeneous fixed points in program analysis, abstract interpretation and semantics [2,6], and fixed point logics. We would also like to compare the expressiveness of heterogeneous fixed points with other fixed point formulations like mu-calculus [16,8], generalized inductive definitions [5], etc.

7 Acknowledgments

Authors wish to thank Patrick Cousot for useful comments and suggestions on a related technical report [10] on heterogeneous fixed points. Authors are thankful to Supratik Chakraborty, Bageshri Sathe, and Amey Karkare for some useful discussions.

References

1. J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245. ACM Press, 1993.
2. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.
3. M. Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, School of Computer Science, McGill University, Montreal, 1993.
4. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256. ACM Press, 1994.
5. S. Feferman. Formal theories for transfinite iterations of generalized inductive definitions and some subsystems of analysis. In A. Kino, J. Myhill, and R. E. Vesley, editors, *Intuitionism and Proof Theory: Proceedings of the Summer Conference at Buffalo, N.Y. Studies in Logic and the Foundations of Mathematics.*, pages 303–326. North-Holland, 1968.
6. R. Giacobazzi and I. Mastroeni. Compositionality in the puzzle of semantics. In *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 87–97. ACM press, 2002.
7. M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, 1999.
8. P. Hitchcock and D. Park. Induction rules and termination proofs. In M. Nivat, editor, *Proceedings 1st Symp. on Automata, Languages, and Programming, ICALP'72, Paris, France, 3–7 July 1972*, pages 225–251. Amsterdam, 1973.

9. A. Kanade, U. Khedker, and A. Sanyal. Equivalence of may-must and definite-possibly points-to analyses. Dept. Computer Science and Engg., Indian Institute of Technology, Bombay, April 2005. `http://www.cse.iitb.ac.in/ aditya/reports/equivalence-points-to.ps`.
10. A. Kanade, A. Sanyal, and U. Khedker. Heterogeneous fixed points. Technical Report TR-CSE-001-05, Dept. Computer Science and Engg., Indian Institute of Technology, Bombay, January 2005. `http://www.cse.iitb.ac.in/ aditya/reports/TR-CSE-001-05.ps`.
11. U. Khedker. *The Compiler Design Handbook: Optimizations and Machine Code Generation*, chapter Data Flow Analysis. CRC Press, 2002.
12. G. A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, October 1973.
13. S. C. Kleene. *Introduction to Mathematics.* D. Van Nostrand, 1952.
14. B. Knaster. Une théorème sur les fonctions d'ensembles. *Annales Soc. Polonaise Math.*, 6:133–134, 1928.
15. J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 24–31. ACM Press, 1988.
16. D. Scott and J. W. de Bakker. A theory of programs. Unpublished notes, IBM seminar, Vienna, 1969.
17. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

A Overview of Emami's Points-To Analysis

Several representations have been proposed for capturing the aliasing information [1,7,15,4]. The points-to analysis [4] due to Emami et al. computes *points-to relation* between pointer expressions denoting stack locations. The aliases that hold along some but *not* along all paths are captured by *possibly* points-to relation. If a variable x possibly contains address of a variable y, then it is denoted by (x, y, P). The aliases that hold along all paths are captured by *definite* points-to relation. If a variable x definitely contains address of a variable y, then it is denoted by (x, y, D).

For simplicity of exposition, we consider a subset of the language in [4]. Non-pointer assignments are ignored. The pointer expressions may consist of scalar and pointer variables, referencing operator '&', and dereferencing operator '$*$'. The left-hand side (lhs) of an assignment can be either x or $*x$ and the right-hand side (rhs) can be x, $\&x$, or $*x$, for some variable x. We restrict ourselves to intraprocedural analysis and view the program as a flow graph. The nodes in the graph are either empty or contain a single pointer assignment.

We now abstract the algorithm in [4] as data flow equations. The points-to relation at IN of a node i is a confluence of points-to relations at OUT of its predecessors $p_1, \ldots, p_k$.

$$\mathsf{input}_i = \begin{cases} \mathsf{Merge}\left(\mathsf{output}_{p_1}, \ldots, \mathsf{output}_{p_k}\right) & \text{if } i \neq \text{entry} \\ \phi & \text{if } i = \text{entry} \end{cases} \tag{22}$$

where "entry" is the unique entry node of the procedure and the operation Merge [3], defined below, is extended to multiple arguments in an obvious way.

$$\begin{aligned}\mathsf{Merge}(S_1, S_2) = &\{(p_1, p_2, D) \mid (p_1, p_2, D) \in S_1 \cap S_2\} \cup \\ &\{(p_1, p_2, P) \mid (p_1, p_2, r) \in S_1 \cup S_2 \wedge (p_1, p_2, D) \notin S_1 \cap S_2\} \quad (23)\end{aligned}$$

Note that the definition of Merge excludes the definite information along all paths from being considered as possible points-to information.

An *R-location* represents the variable whose address appears in the rhs. An *L-location* represents the variable which is being assigned this address. Both of the L-location and R-location depend on the nature of points-to information and could be either *definite* or *possible*. Let L_i denote the set of L-locations of an assignment in node i tagged with D or P appropriately. Let R_i denote the corresponding R-locations. Then,

lhs_i	L_i
x	$\{(x, D)\}$
$*x$	$\{(y, r) \mid (x, y, r) \in \mathsf{input}_i\}$

rhs_i	R_i
$\&x$	$\{(x, D)\}$
x	$\{(y, r) \mid (x, y, r) \in \mathsf{input}_i\}$
$*x$	$\{(z, r_1 \oplus r_2) \mid (x, y, r_1), (y, z, r_2) \in \mathsf{input}_i\}$

where lhs_i and rhs_i are lhs and rhs of the assignment in node i, and $\oplus$ is defined as

$$r_1 \oplus r_2 \triangleq \begin{cases} P & \text{if } r_1 \neq r_2 \\ r_1 & \text{otherwise} \end{cases}$$

Let (x, y, r) hold before an assignment in node i.

– If $(x, P) \in \mathsf{L}_i$, then x may or may not be modified. Thus after the assignment, x may or may not point to y. Hence the definiteness of its pointing to y must be changed to the possibility of its pointing to y. This is captured by (24) and (25) below.
– If $(x, D) \in \mathsf{L}_i$, then x is definitely modified as a side effect of the assignment and x ceases to point to y. This is captured by (26) below.

$$\mathsf{change\_set}_i = \{(x, y, D) \mid \underline{(x, P) \in \mathsf{L}_i} \wedge (x, y, D) \in \mathsf{input}_i\} \quad (24)$$

$$\mathsf{changed\_input}_i = (\mathsf{input}_i - \mathsf{change\_set}_i) \cup \{(x, y, P) \mid (x, y, D) \in \mathsf{change\_set}_i\} \quad (25)$$

$$\mathsf{kill\_set}_i = \{(x, y, r) \mid \underline{(x, D) \in \mathsf{L}_i} \wedge (x, y, r) \in \mathsf{input}_i\} \quad (26)$$

An assignment generates definite points-to information between definite L-locations and definite R-locations. All other combinations between L-locations and R-locations are generated as possibly points-to information.

$$\begin{aligned}\mathsf{gen\_set}_i = &\{(x, y, D) \mid (x, D) \in \mathsf{L}_i \;\wedge\; (y, D) \in \mathsf{R}_i\} \cup \\ &\{(x, y, P) \mid (x, r_1) \in \mathsf{L}_i \wedge (y, r_1) \in \mathsf{R}_i \wedge (r_1 \neq D \vee r_2 \neq D)\} \quad (27)\end{aligned}$$

Finally, the points-to information at OUT of node i is

$$\mathsf{output}_i = (\mathsf{changed\_input}_i - \mathsf{kill\_set}_i) \cup \mathsf{gen\_set}_i \quad (28)$$

Register Allocation Via Coloring of Chordal Graphs

Fernando Magno Quintão Pereira and Jens Palsberg

UCLA Computer Science Department,
University of California, Los Angeles

Abstract. We present a simple algorithm for register allocation which is competitive with the iterated register coalescing algorithm of George and Appel. We base our algorithm on the observation that 95% of the methods in the Java 1.5 library have chordal interference graphs when compiled with the JoeQ compiler. A greedy algorithm can optimally color a chordal graph in time linear in the number of edges, and we can easily add powerful heuristics for spilling and coalescing. Our experiments show that the new algorithm produces better results than iterated register coalescing for settings with few registers and comparable results for settings with many registers.

1 Introduction

Register allocation is one of the oldest and most studied research topics of computer science. The goal of register allocation is to allocate a finite number of machine registers to an unbounded number of temporary variables such that temporary variables with interfering live ranges are assigned different registers. Most approaches to register allocation have been based on graph coloring. The graph coloring problem can be stated as follows: given a graph G and a positive integer K, assign a color to each vertex of G, using at most K colors, such that no two adjacent vertices receive the same color. We can map a program to a graph in which each node represents a temporary variable and edges connect temporaries whose live ranges interfere. We can then use a coloring algorithm to perform register allocation by representing colors with machine registers.

In 1982 Chaitin [8] reduced graph coloring, a well-known NP-complete problem [18], to register allocation, thereby proving that also register allocation is NP-complete. The core of Chaitin's proof shows that the interference relations between temporary variables can form any possible graph. Some algorithms for register allocation use integer linear programming and may run in worst-case exponential time, such as the algorithm of Appel and George [2]. Other algorithms use polynomial-time heuristics, such as the algorithm of Briggs, Cooper, and Torczon [5], the Iterated Register Coalescing algorithm of George and Appel [12], and the Linear Scan algorithm of Poletto and Sarkar [16]. Among the polynomial-time algorithms, the best in terms of resulting code quality appears to be iterated register coalescing. The high quality comes at the price of handling spilling and coalescing of temporary variables in a complex way. Figure 1

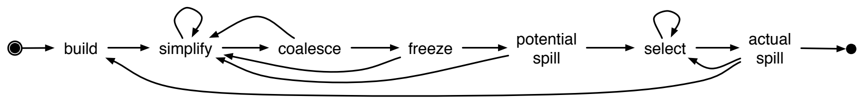


Fig. 1. The iterated register coalescing algorithm

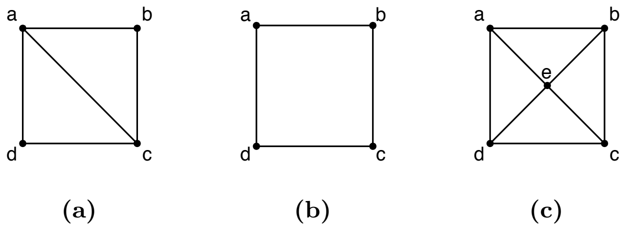


Fig. 2. (a) A chordal graph. (b-c) Two non-chordal graphs.

illustrates the complexity of iterated register coalescing by depicting the main phases and complicated pattern of iterations of the algorithm. In this paper we show how to design algorithms for register allocation that are simple, efficient, and competitive with iterated register coalescing.

We have observed that the interference graphs of real-life programs tend to be *chordal* graphs. For example, 95% of the methods in the Java 1.5 library have chordal interference graphs when compiled with the JoeQ compiler. A graph is chordal if every cycle with four or more edges has a *chord*, that is, an edge which is not part of the cycle but which connects two vertices on the cycle. (Chordal graphs are also known as 'triangulated', 'rigid-circuit', 'monotone transitive', and 'perfect elimination' graphs.) The graph in Figure 2(a) is chordal because the edge *ac* is a chord in the cycle *abcda*. The graph in Figure 2(b) is non-chordal because the cycle *abcda* is chordless. Finally, the graph in Figure 2(c) is non-chordal because the cycle *abcda* is chordless, just like in Figure 2(b).

Chordal graphs have several useful properties. Problems such as *minimum coloring*, *maximum clique*, *maximum independent set* and *minimum covering by cliques*, which are NP-complete in general, can be solved in polynomial time for chordal graphs [11]. In particular, optimal coloring of a chordal graph $G = (V, E)$ can be done in $O(|E| + |V|)$ time.

In this paper we present an algorithm for register allocation, which is based on a coloring algorithm for chordal graphs, and which contains powerful heuristics for spilling and coalescing. Our algorithm is simple, efficient, and modular, and it performs as well, or better, than iterated register coalescing on both chordal graphs and non-chordal graphs.

The remainder of the paper is organized as follows: Section 1 discusses related work, Section 3 summarizes some known properties and algorithms for chordal graphs, Section 4 describes our new algorithm, Section 5 presents our experimental results, and Section 6 concludes the paper.

2 Related Work

We will discuss two recent efforts to design algorithms for register allocation that take advantage of properties of the underlying interference graphs. Those efforts center around the notions of perfect and 1-perfect graphs. In a 1-perfect graph, the chromatic number, that is, the minimum number of colors necessary to color the graph, equals the size of the largest clique. A perfect graph is a 1-perfect graph with the additional property that every induced subgraph is 1-perfect. Every chordal graph is perfect, and every perfect graph is 1-perfect.

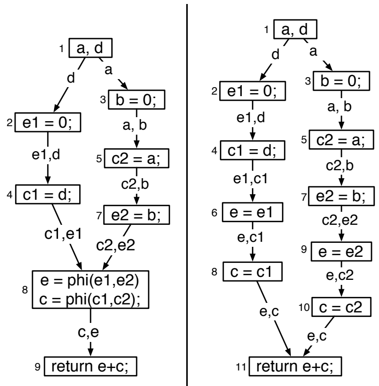
Andersson [1] observed that all the 27,921 interference graphs made publicly available by George and Appel [3] are 1-perfect, and we have further observed that 95.6% of those graphs are chordal when the interferences between precolored registers and temporaries are not considered. Andersson also showed that an optimal, worst-case exponential time algorithm for coloring 1-perfect graphs is faster than iterated register coalescing when run on those graphs.

Recently, Brisk et al. [6] proved that *strict* programs in SSA-form have *perfect* interference graphs; independently, Hack [14] proved the stronger result that strict programs in SSA-form have *chordal* interference graphs. A strict program [7] is one in which every path from the initial block until the use of a variable v passes through a definition of v. Although perfect and chordal graphs can be colored in polynomial time, the practical consequences of Brisk and Hack's proofs must be further studied. SSA form uses a notational abstraction called *phi-function*, which is not implemented directly but rather replaced by copy instructions during an SSA-elimination phase of the compiler. Register allocation after SSA elimination is NP-complete [15].

For example, Figure 3(a) shows a program with a non-chordal interference graph, Figure 3(b) shows the program in SSA form, and Figure 3(c) shows the

```
int m(int x, a, d) {
    int b, c;
    if(x > 0) {
        e = 0;
        c = d;
    } else {
        b = 0;
        c = a;
        e = b;
    }
    return e + c;
}
```

(a)



(b) **(c)**

Fig. 3. (a) A program with a non-chordal interference graph, (b) the program in SSA form, (c) the program after SSA elimination program after SSA elimination. The example program in Figure 3(a) has a cycle of five nodes without chords: a–d–e–c–b–a. In the example in Figure 3(b), $e = phi(e_1, e_2)$ will return e_2 if control reaches block 8 through block 7, and will return e_1 if control reaches block 8 through block 4. The SSA semantics states that all phi-functions at the beginning of a block must be evaluated simultaneously as the first operation upon entering that block; thus, live ranges that reach block 8 do not interfere with live ranges that leave block 8. Hack [14] used this observation to show that phi-functions break chordless cycles so strict programs in SSA-form have chordal interference graphs. The example program after SSA elimination, in Figure 3(c), has an interference graph which is non-chordal, non-perfect, and even non-1-perfect: the largest clique has two nodes but three colors are needed to color the graph. Note that the interference graph has a cycle of seven nodes without chords: a–d–e1–c1–e–c2–b–a.

For 1-perfect graphs, recognition and coloring are NP-complete. Perfect graphs can be recognized and colored in polynomial time, but the algorithms are highly complex. The recognition of perfect graphs is in $O(|V|^9)$ time [9]; the complexity of the published coloring algorithm [13] has not been estimated accurately yet. In contrast, chordal graphs can be recognized and colored in $O(|E| + |V|)$ time, and the algorithms are remarkably simple, as we discuss next.

3 Chordal Graphs

We now summarize some known properties and algorithms for chordal graphs. For a graph G, we will use $\Delta(G)$ to denote the maximum outdegree of any vertex in G, and we will use $N(v)$ to denote the set of neighbors of v, that is, the set of vertices adjacent to v in G. A *clique* in an undirected graph $G = (V, E)$ is a subgraph in which every two vertices are adjacent. A vertex $v \in V$ is called *simplicial* if its neighborhood in G is a clique. A *Simplicial Elimination Ordering* of G is a bijection $\sigma : V(G) \rightarrow \{1 \dots |V|\}$, such that every vertex v_i is a simplicial vertex in the subgraph induced by $\{v_1, \dots, v_i\}$. For example, the vertices b, d of the graph shown in Figure 2(a) are simplicial. However, the vertices a and c are not, because b and d are not connected. In this graph, $\langle b, a, c, d \rangle$ is a simplicial elimination ordering. There is no simplicial elimination ordering ending in the nodes a or c. The graphs depicted in Figures 2(b) and 2(c) have no simplicial elimination orderings.

Theorem 1. (Dirac [10]) *An undirected graph without self-loops is chordal if and only if it has a simplicial elimination ordering.*

The algorithm *greedy coloring*, outlined in Figure 4, is a $O(E)$ heuristic for graph coloring. Given a graph G and a sequence of vertices ν, *greedy coloring* assigns to each vertex of ν the next available color. Each color is a number c where $0 \leq c \leq \Delta(G) + 1$. If we give *greedy coloring* a simplicial elimination ordering of the vertices, then the greedy algorithm yields an optimal coloring [11]. In other words, *greedy coloring* is optimal for chordal graphs.

```
procedure greedy coloring
1    input: $G = (V, E)$, a sequence of vertices $\nu$
2    output: a mapping $m$, $m(v) = c, 0 \leq c \leq \Delta(G) + 1, v \in V$
3    For all $v \in \nu$ do $m(v) \leftarrow \perp$
4    For $i \leftarrow 1$ to $|\nu|$ do
5        let $c$ be the lowest color not used in $N(\nu(i))$ in
6            $m(\nu(i)) \leftarrow c$
```

Fig. 4. The greedy coloring algorithm

```
procedure MCS
1    input: $G = (V, E)$
2    output: a simplicial elimination ordering $\sigma = v_1, \ldots, v_n$
3    For all $v \in V$ do $\lambda(v) \leftarrow 0$
4    For $i \leftarrow 1$ to $|V|$ do
5        let $v \in V$ be a vertex such that $\forall u \in V, \lambda(v) \geq \lambda(u)$ in
6            $\sigma(i) \leftarrow v$
7            For all $u \in V \cap N(v)$ do $\lambda(u) \leftarrow \lambda(u) + 1$
8            $V \leftarrow V - \{v\}$
```

Fig. 5. The maximum cardinality search algorithm

The algorithm known as *Maximum Cardinality Search* (MCS)[17] recognizes and determines a simplicial elimination ordering σ of a chordal graph in $O(|E| + |V|)$ time. MCS associates with each vertex v of G a weight $\lambda(v)$, which initially is 0. At each stage MCS adds to σ the vertex v of greatest weight not yet visited. Subsequently MCS increases by one the weight of the neighbors of v, and starts a new phase. Figure 5 shows a version of MCS due to Berry et al. [4].

The procedure MCS can be implemented to run in $O(|V| + |E|)$ time. To see that, notice that the first loop executes $|V|$ iterations. In the second loop, for each vertex of G, all its neighbors are visited. After a vertex is evaluated, it is removed from the remaining graph. Therefore, the weight λ is increased exactly $|E|$ times. By keeping vertices in an array of buckets indexed by λ, the vertex of highest weight can be found in $O(1)$ time.

4 Our Algorithm

Our algorithm has several independent phases, as illustrated in Figure 6, namely coloring, spilling, and coalescing, plus an optional phase called *pre-spilling*. Coalescing must be the last stage in order to preserve the optimality of the coloring algorithm, because, after merging nodes, the resulting interference graph can be non-chordal. Our algorithm uses the *MCS* procedure (Figure 5) to produce an ordering of the nodes, for use by the pre-spilling and coloring phases. Our approach yields optimal colorings for chordal graphs, and, as we show in Section 5, it produces competitive results even for non-chordal graphs. We have

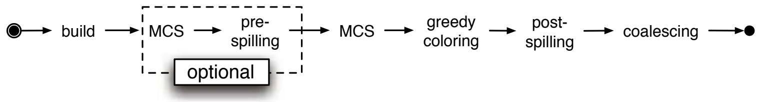
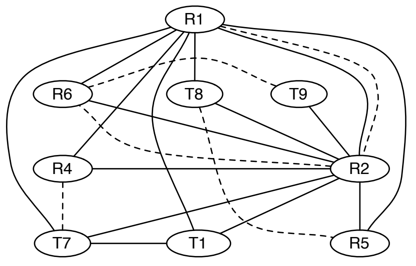


Fig. 6. The main phases of our algorithm

```
int gcd (int R1, int R2)
1.  IFCMP_I_EQ    ..  R2   0   (12);
2.  ZERO_CHECK_I  ..  T1  R2;
3.  DIV_I         T7  R1   R2;
4.  CHECK_EX      T1
5.  MOVE_I        R4  T7;
6.  MUL_I         T8  R2   R4;
7.  MOVE_I        R5  T8;
8.  SUB_I         T9  R1   R5;
9.  MOVE_I        R6  T9;
10. MOVE_I        R1  R2;
11. MOVE_I        R2  R6;
12. GOTO          ..  ..   ..  (1);
13. RETURN_I      ..  R1;
```

(a)



(b)

Fig. 7. (a) Euclid's algorithm. (b) Interference graph generated for gcd().

implemented heuristics, rather than optimal algorithms, for spilling and coalescing. Our experimental results show that our heuristics perform better than those used in the iterated register coalescing algorithm.

In order to illustrate the basic principles underlying our algorithm, we will as a running example show how our algorithm allocates registers for the program in Figure 7 (a). This program calculates the greatest common divisor between two integer numbers using Euclid's algorithm. In the intermediate representation adopted, instructions have the form op, t, p_1, p_2. Such an instruction defines the variable t, and adds the temporaries p_1 and p_2 to the chain of used values. The interference graph yielded by the example program is shown in Figure 7 (b). Solid lines connecting two temporaries indicate that they are simultaneously alive at some point in the program, and must be allocated to different registers. Dashed lines connect move related registers.

Greedy Coloring. In order to assign machine registers to variables, the *greedy coloring* procedure of Figure 4 is fed with an ordering of the vertices of the interference graph, as produced by the *MCS* procedure. From the graph shown in Figure 7 (b), MCS produces the ordering: ⟨ `T7, R1, R2, T1, R5, R4, T8, R6, T9` ⟩, and *greedy coloring* then produces the mapping between temporaries and colors that is outlined in Figure 8 (a). If the interference graph is chordal, then the combination of *MCS* and *Greedy Coloring* produces a minimal coloring. The coloring phase uses an unbounded number of colors so that the interference graph can always be colored. The excess of colors will be removed in the post-spilling stage.

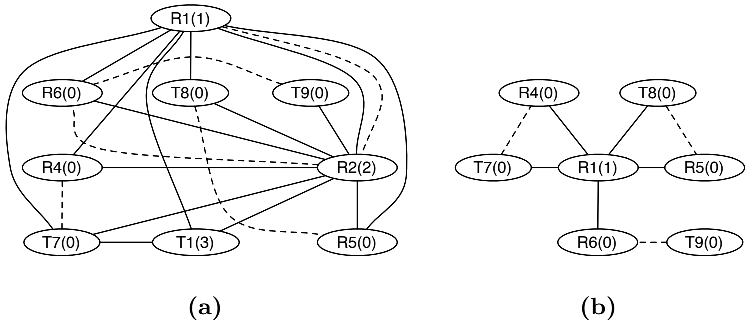


Fig. 8. (a) Colored interference graph. (b) Interference graph after spilling the highest colors.

Post-Spilling. Given an instance of a register allocation problem, it may be possible that the number of available registers is not sufficient to accommodate all the temporary variables. In this case, temporaries must be removed until the remaining variables can be assigned to registers. The process of removing temporaries is called spilling. A natural question concerning spilling when the interference graph is chordal is if there is a polynomial algorithm to determine the minimum number of spills. The problem of determining the maximum K-colorable subgraph of a chordal graph is NP-complete [20], but has polynomial solution when the number of colors (K) is fixed. We do not adopt the polynomial algorithm because its complexity seems prohibitive, namely $O(|V|^K)$ time.

Iterated register coalescing performs spilling as an iterative process. After an unsuccessful attempt to color the interference graph, some vertices are removed, and a new coloring phase is executed. We propose to spill nodes in a single iteration, by removing in each step all nodes of a chosen color from the colored interference graph. The idea is that given a K-colored graph, if all the vertices sharing a certain color are removed, the resulting subgraph can be colored with $K - 1$ colors. We propose two different heuristics for choosing the next color to be removed: (i) remove the least-used color, and (ii) remove the highest color assigned by the greedy algorithm.

The spilling of the highest color has a simpler and more efficient implementation. The heuristic is based on the observation that the greedy coloring tends to use the lower colors first. For a chordal graph, the number of times the highest color is used is bounded by the number of maximal cliques in the interference graph. A maximal clique is a clique that cannot be augmented. In other words, given a graph $G = (V, E)$, a clique Q is maximal if there is no vertex $v, v \in V - Q$, such that v is adjacent to all the vertices of Q. For our running example, Figure 8 (b) shows the colored interference graph after the highest colors have been removed, assuming that only two registers are available in the target machine. Coincidentally, the highest colors are also the least-used ones.

```
procedure coalescing
1   input: list $l$ of copy instructions, $G = (V, E)$, $K$
2   output: $G'$, the coalesced graph $G$
3   let $G' = G$ in
4   for all $x := y \in l$ do
5       let $S_x$ be the set of colors in $N(x)$
6       let $S_y$ be the set of colors in $N(y)$
7       if there exists $c, c < K, c \notin S_x \cup S_y$ then
8           let $xy, xy \notin V$ be a new node
9           add $xy$ to $G'$ with color $c$
10          make $xy$ adjacent to every $v, v \in N(x) \cup N(y)$
11          replace occurrences of $x$ or $y$ in $l$ by $xy$
12          remove $x$ from $G'$
13          remove $y$ from $G'$
```

Fig. 9. The greedy coalescing algorithm

Coalescing. The last phase of the algorithm is the coalescing of move related instructions. Coalescing helps a compiler to avoid generating redundant copy instructions. Our coalescing phase is executed in a greedy fashion. For each instruction $a := b$, the algorithm looks for a color c not used in $N(a) \cup N(b)$, where $N(v)$ is the set of neighbors of v. If such a color exists, then the temporaries a and b are coalesced into a single register with the color c. This algorithm is described in Figure 9. Our current coalescing algorithm does not use properties of chordal graphs; however, as future work, we plan to study how coalescing can take benefit from chordality.

Pre-Spilling. To color a graph, we need a number of colors which is at least the size of the largest clique. We now present an approach to removing nodes that will bring the size of the largest clique down to the number of available colors and guarantee that the resulting graph will be colorable with the number of available colors (Theorem 2). Gavril [11] has presented an algorithm *maximalCl*, shown in Figure 10, which lists all the maximal cliques of a chordal graph in $O(|E|)$ time. Our pre-spilling phase first runs *maximalCl* and then the procedure *pre-spilling* shown in Figure 11. Pre-spilling uses a map ω which maps each vertex to an approximation of the number of maximal cliques that contain that vertex. The objective of pre-spilling is to minimize the number of spills. When an interference graph is non-chordal, the *maximalCl* algorithm may return graphs that are not all cliques and so pre-spilling may produce unnecessary spills. Nevertheless, our experimental results in Section 5 show that the number of spills is competitive even for non-chordal graphs.

The main loop of pre-spilling performs two actions: (i) compute the vertex v that appears in most of the cliques of ξ and (ii) remove v from the cliques in which it appears. In order to build an efficient implementation of the pre-spilling algorithm, it is helpful to define a bidirectional mapping between vertices and the cliques in which they appear. Because the number of maximal cliques is

```
procedure maximalCl
1   input: $G = (V, E)$
2   output: a list of cliques $\xi = \langle Q_1, Q_2, \ldots, Q_n \rangle$
3   $\sigma \leftarrow$ MCS(G)
4   For $i \leftarrow 1$ to $n$ do
5       Let $v \leftarrow \sigma[i]$ in
6           $Q_i \leftarrow \{v\} \cup \{u \mid (u, v) \in E, u \in \{\sigma[1], \ldots, \sigma[i-1]\}\}$
```

Fig. 10. Listing maximal cliques in chordal graphs

```
procedure pre-spilling
1   input: $G = (V, E)$, a list of subgraphs of $G$: $\xi = \langle Q_1, Q_2, \ldots, Q_n \rangle$,
        a number of available colors $K$, a mapping $\omega$
2   output: a $K$-colorable subgraph of $G$
3   $R_1 = Q_1; R_2 = Q_2; \ldots R_n = Q_n$
4   while there is $R_i$ with more than $K$ nodes do
5       let $v \in R_i$ be a vertex such that $\forall u \in R_i, \omega(v) \geq \omega(u)$ in
6           remove $v$ from all the graphs $R_1, R_2, \ldots, R_n$
7   return $R_1 \cup R_2 \cup \ldots \cup R_n$
```

Fig. 11. Spilling intersections between maximal cliques

bounded by $|V|$ for a chordal graph, it is possible to use a bucket list to compute $\omega(v), v \in V$ in $O(1)$ time. After a temporary is deleted, a number of cliques may become K-colorable, and must be removed from ξ. Again, due to the bidirectional mapping between cliques and temporaries, this operation can be performed in $O(|N(v)|)$, where $N(v)$ is the set of vertices adjacent to v. Overall, the spilling algorithm can be implemented in $O(|E|)$.

Theorem 2. *The graph pre-spilling(G,maximalCl(G),K,ω) is K-colorable.*

Proof. Let $\langle Q_1, Q_2, \ldots, Q_n \rangle$ be the output of *maximalCl*(G). Let $R_1 \cup R_2 \cup \ldots \cup R_n$ be the output of *pre-spilling*$(G$,*maximalCl*$(G),K,\omega)$. Let $R_i^\bullet = R_1 \cup R_2 \cup \ldots \cup R_i$ for $i \in 1..n$.

We will show that for all $i \in 1..n$, $R_i^\bullet$ is K-colorable. We proceed by induction on i.

In the base case of $i = 1$, we have $R_1^\bullet = R_1 \subseteq Q_1$ and Q_1 has exactly one node. We conclude that $R_1^\bullet$ is K-colorable.

In the induction step we have from the induction hypothesis that $R_i^\bullet$ is K-colorable so let c be a K-coloring of $R_i^\bullet$. Let v be the node $\sigma[i+1]$ chosen in line 5 of *maximalCl*. Notice that v is the only vertex of Q_{i+1} that does not appear in $Q_1, Q_2, \ldots, Q_i$ so c does not assign a color to v. Now there are two cases. First, if v has been removed by *pre-spilling*, then $R_{i+1}^\bullet = R_i^\bullet$ so c is a K-coloring of $R_{i+1}^\bullet$. Second, if v has not been removed by *pre-spilling*, then we use that R_{i+1} has at most K nodes to conclude that the degree of v in R_{i+1} is at most $K - 1$. We have that c assigns a color to all neighbors for v in R_{i+1} so we have a color left to assign to v and can extend c to a K-coloring of $R_{i+1}^\bullet$.

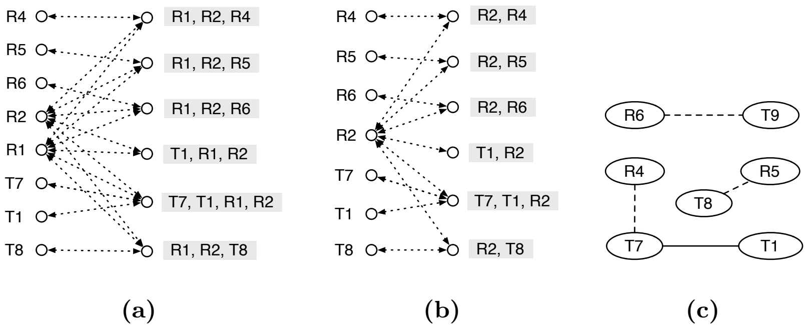


Fig. 12. (a) Mapping between nodes and maximal cliques. (b) Mapping after pruning node `R1`. (c) Interference graph after spilling `R1` and `R2`.

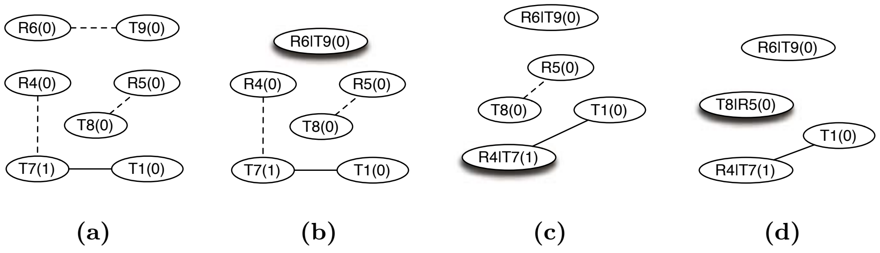


Fig. 13. (a) Coloring produced by the greedy algorithm. (b) Coalescing `R6` and `T9`. (c) Coalescing `R4` and `T7`. (d) Coalescing `R5` and `T8`.

Figure 12 (a) shows the mapping between temporaries and maximal cliques that is obtained from the `gcd(x, y)` method, described in Figure 7 (a). Assuming that the target architecture has two registers, the cliques must be pruned until only cliques of size less than two remain. The registers `R1` and `R2` are the most common in the maximal cliques, and, therefore, should be deleted. The configuration after removing register `R1` is outlined in Figure 12 (b). After the pruning step, all the cliques are removed from ξ. Figure 12 (c) shows the interference graph after the spilling phase.

Figure 13 outlines the three possible coalescings in this example. Coincidentally, two of the move related registers were assigned the same color in the greedy coloring phase. Because of this, their colors do not had to be changed during the coalescing stage. The only exception is the pair (`R4, T7`). In the coalescing phase, the original color of `R4` is changed to the same color of `T7`. Afterwards, the registers are merged.

Complexity Analysis. The coloring phase, as a direct application of maximum cardinality search and greedy coloring, can be implemented to run in $O(|V|+|E|)$ time.

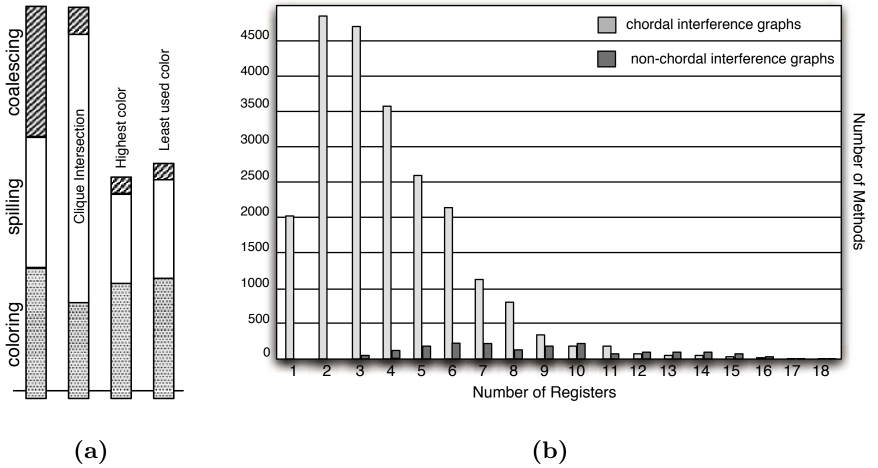


Fig. 14. (a) Time spent on coloring, spilling and coalescing in the different heuristics. (b) Number of registers assigned to methods of the Java 1.5 Standard Library.

Our heuristics for spilling can all can be implemented to run in $O(|E|)$ time. In order to implement spilling of the least-used color, it is possible to order the colors with bucket sort, because the maximum color is bounded by the highest degree of the interference graph plus one. The same technique can be used to order the weight function for the pre-spilling algorithm because the size of the list ξ, produced by the procedure *maximalCl*, is bounded by $|V|$.

Coalescing is the phase with the highest complexity, namely $O(t^3)$, where t is the number of temporaries in the source code. Our coalescing algorithm inspects, for each pair of move related instructions, all their neighbors. It is theoretically possible to have up to t^2 pairs of move related instructions in the target code. However, the number of these instructions is normally small, and our experimental results show that the coalescing step accounts for less than 10% of the total running time (see Figure 14 (a)).

5 Experimental Results

We have built an evaluation framework in Java, using the JoeQ compiler [19], in order to compare our algorithm against the iterated register coalescing. When pre-spilling is used, post-spilling is not necessary (Theorem 2). Our benchmark suite is the entire run-time library of the standard Java 1.5 distribution, i.e. the set of classes in `rt.jar`. In total, we analyzed 23,681 methods. We analyzed two different versions of the target code. One of them is constituted by the intermediate representation generated by JoeQ without any optimization. In the other version, the programs are first converted to single static assignment form (SSA), and them converted back to the JoeQ intermediate representation, by

Table 1. Comparison between our algorithm (NIA) and Iterated Register Coalescing (IRC), including results for the three different spilling heuristics in Section 4

Algorithm	SSA	number of registers	register/ method	spill/ method	Total spills	maximum # spills	coalescing/ moves	running time (s)
NIA	no	18	4.20	0.0044	102	15	0.38	2645.1
Post-spilling	yes	18	4.13	0.0034	81	14	0.72	2769.9
least-used	no	6	3.79	0.43	10,218	30	0.37	2645.0
color	yes	6	3.75	0.51	12,108	91	0.73	2781.7
NIA	no	18	4.20	0.0048	115	15	0.34	2641.5
Post-spilling	yes	18	4.13	0.010	246	63	0.72	2767.0
highest	no	6	3.80	0.50	11,923	33	0.35	2674.3
used color	yes	6	3.75	0.80	19,018	143	0.69	2764.2
NIA	no	18	4.20	0.0044	105	15	0.34	2640.5
Pre-spilling	yes	18	4.13	0.0039	94	17	0.72	2763.2
	no	6	3.78	0.45	10,749	34	0.35	2645.8
	yes	6	3.75	0.49	11,838	43	0.70	2765.1
IRC	no	18	4.25	0.0050	115	16	0.31	2644.1
	yes	18	4.17	0.0048	118	27	0.70	2823.2
	no	6	3.81	0.50	11,869	32	0.31	2641.5
	yes	6	3.77	0.57	13,651	86	0.66	2883.7

substituting the *phi* functions by copy instructions. In the former case, approximately 91% of the interference graphs produced are chordal. In the latter, the percentage of chordal graphs is 95.5%.

Table 1 shows results obtained by the iterative algorithm (IRC), and our non-iterative register allocator (NIA). The implementation of both algorithms attempts to spill the minimum number of registers. As it can be seen in the table, our technique gives better results than the traditional register allocator. It tends to use less registers per method, because it can find an optimum assignment whenever the interference graph is chordal. Also, it tends to spill less temporaries, because, by removing intersections among cliques, it decreases the chromatic number of several clusters of interfering variables at the same time. Notably, for the method `coerceData`, of the class `java.awt.image.ComponentColorModel`, with 6 registers available for allocation, the pre-spilling caused the eviction of 41 temporaries, whereas Iterated Register Coalescing spilled 86. Also, because our algorithm tends to spill fewer temporaries and to use fewer registers in the allocation, it is able to find more opportunities for coalescing. The Iterated register coalescing and our algorithm have similar running times. The complexity of a single iteration of the IRC is $O(|E|)$, and the maximum number of iterations observed in the tests was 4; thus, its running time can be characterized as linear. Furthermore, both algorithms can execute a cubic number of coalescings, but, in the average, the quantity of copy instructions per program is small when compared to the total number of instructions.

Table 2 compares the two algorithms when the interference graphs are chordal and non-chordal. This data refers only to target programs after SSA elimination.

Table 2. Comparative performance of our spilling heuristics for chordal and non-chordal interference graphs

Algorithm	chordal graph	number of registers	register/ method	spill/ method	Total spills	maximum # spills	coalescing/ moves
NIA	no	18	8.17	0.054	61	17	0.75
Pre-spilling	no	6	5.77	4.55	5173	43	0.79
	yes	18	3.92	0.0015	33	6	0.69
	yes	6	3.65	0.29	6665	31	0.68
IRC	no	18	8.39	0.062	71	27	0.74
	no	6	5.79	4.89	5562	86	0.66
	yes	18	3.97	0.0015	34	6	0.67
	yes	6	3.68	0.39	8089	45	0.67

Table 3. Results obtained from the allocation of registers to 27,921 interference graphs generated from ML code

Algorithm	chordal graph	Total of spills	maximum number of spills	coalescing/ moves	allocation time (s)
Post-spilling least used color	yes	1,217	84	0.97	223.8
	no	63	14	0.94	
Post-spilling highest used color	yes	1,778	208	0.97	222.9
	no	80	20	0.94	
Pre-spilling	yes	1,127	86	0.97	482.3
	no	1,491	23	0.93	

In general, non-chordal interference graphs are produced by complex methods. For instance, methods whose interference graphs are non-chordal use, on average, 80.45 temporaries, whereas the average for chordal interference graphs is 13.94 temporaries.

The analysis of methods whose interference graphs are chordal gives some insight about the structure of Java programs. When an interference graph is chordal, the mapping between temporaries and registers is optimal, i.e. it uses the smallest possible number of registers. Figure 14 (b) shows the relation between number of methods of the Java Library and the minimum number of registers necessary to handle them. Only methods that could be colored with less than 18 colors (99.6%) are shown. Allocation results for methods whose interference graph are non-chordal are also presented, even though these may not be optimal.

Figure 14 (a) compares the amount of time spent on each phase of the algorithm when different spilling heuristics are adopted. The time used in the allocation process is a small percentage of the total running time presented in Table 1 because the latter includes the loading of class files, the parsing of bytecodes, the liveness analysis and the construction of the interference graph. When pre-spilling is used, it accounts for more than half the allocation time.

We have also tested our register allocation algorithm on the 27,921 interference graphs published by George and Appel. Those graphs were generated by the standard ML compiler of New Jersey compiling itself [3]. Our tests have shown that 95.7% of the interference graphs are chordal when the interferences between pre-colored registers and temporaries are not taken into consideration. The compilation results are outlined in Table 3. The graphs contain 21 pairwise interfering pre-colored registers, which represent the machine registers available for the allocation. Because of these cliques, all the graphs, after spilling, demanded exactly 21 colors. When the graphs are chordal, pre-spilling gives the best results; however, this heuristic suffers a penalty when dealing with the non-chordal graphs, because they present a 21-clique, and must be colored with 21 registers. In such circumstances, the procedure *maximalCl* from Figure 10 have listed some false maximal cliques, and unnecessary spills have been caused. Overall, the spilling of the least-used colors gives the best results. The execution times for analyzing the ML-compiler-based benchmarks are faster than those for analyzing the Java Library because the latter set of timings includes the times to construct the interference graphs.

6 Conclusion

This paper has presented a non-iterative algorithm for register allocation based on the coloring of chordal graphs. Chordal graphs present an elegant structure and can be optimally colored in $O(|V| + |E|)$ time. For the register allocation problem, we can find an optimal allocation in time linear in the number of interferences between live ranges, whenever the interference graph is chordal. Additionally, our algorithm is competitive even when performing register allocation on non-chordal inputs.

In order to validate the algorithm, we compared it to iterated register coalescing. Our algorithm allocates fewer registers per method and spills fewer temporaries. In addition, our algorithm can coalesce about the same proportion of copy instructions as iterated register coalescing.

In addition to being efficient, our algorithm is modular and flexible. Because it is non-iterative, it presents a simpler design than traditional algorithms based on graph coloring. The spill of temporaries can happen before or after the coloring phase. By performing spilling before coloring, it is possible to assign different weights to temporaries in order to generate better code. Our implementation and a set of interference graphs generated from the Java methods tested can be found at `http://compilers.cs.ucla.edu/fernando/projects/`.

Acknowledgments. We thank Ben Titzer and the reviewers for helpful comments on a draft of the paper. Fernando Pereira is sponsored by the Brazilian Ministry of Education under grant number 218603-9. We were supported by the National Science Foundation award number 0401691.

References

1. Christian Andersson. Register allocation by optimal graph coloring. In *12th Conference on Compiler Construction*, pages 34–45. Springer, 2003.
2. Andrew W Appel and Lal George. Optimal spilling for cisc machines with few registers. In *International Conference on Programming Languages Design and Implementation*, pages 243–253. ACM Press, 2001.
3. Andrew W Appel and Lal George. 27,921 actual register-interference graphs generated by standard ML of New Jersey, version 1.09–`http://www.cs.princeton.edu/~appel/graphdata/`, 2005.
4. Anne Berry, Jean Blair, Pinar Heggernes, and Barry Peyton. Maximum cardinality search for computing minimal triangulations of graphs. *Algorithmica*, 39(4):287–298, 2004.
5. Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, 1994.
6. Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial-time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*. ACM Press, 2005.
7. Zoran Budimlic, Keith D Cooper, Timothy J Harvey, Ken Kennedy, Timothy S Oberg, and Steven W Reeves. Fast copy coalescing and live-range identification. In *International Conference on Programming Languages Design and Implementation*, pages 25–32. ACM Press, 2002.
8. G J Chaitin. Register allocation and spilling via graph coloring. *Symposium on Compiler Construction*, 17(6):98–105, 1982.
9. Maria Chudnovsky, Gerard Cornuejols, Xinming Liu, Paul Seymour, and Kristina Vuskovic. Recognizing berge graphs. *Combinatorica*, 25:143–186, 2005.
10. G A Dirac. On rigid circuit graphs. In *Abhandlungen aus dem Mathematischen Seminar der Universiat Hamburg*, volume 25, pages 71–75. University of Hamburg, 1961.
11. Fanica Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SICOMP*, 1(2):180–187, 1972.
12. Lal George and Andrew W Appel. Iterated register coalescing. *Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):300–324, 1996.
13. M Grotschel, L Lovasz, and A Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.
14. Sebastian Hack. Interference graphs of programs in SSA-form. Technical report, Universitat Karlsruhe, 2005.
15. Fernando M Q Pereira and Jens Palsberg. Register allocation after SSA elimination is NP-complete. Manuscript, 2005.
16. Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.
17. Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.
18. Douglas B West. *Introduction to Graph Theory*. Prentice Hall, 2nd edition, 2001.
19. John Whaley. Joeq:a virtual machine and compiler infrastructure. In *Workshop on Interpreters, virtual machines and emulators*, pages 58–66. ACM Press, 2003.
20. Mihalis Yannakakis and Fanica Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, 24(2):133–137, 1987.

Transformation to Dynamic Single Assignment Using a Simple Data Flow Analysis[*]

Peter Vanbroekhoven[1,**], Gerda Janssens[1], Maurice Bruynooghe[1], and Francky Catthoor[2]

[1] Katholieke Universiteit Leuven, Belgium
[2] Interuniversity MicroElectronics Center, Belgium

Abstract. This paper presents a novel method to construct a dynamic single assignment (DSA) form of array-intensive, pointer-free C programs (or in any other procedural language). A program in DSA form does not perform any destructive update of scalars and array elements, *i.e.*, each element is written at most once. As DSA makes the dependencies between variable references explicit, it facilitates complex analyses and optimizations of programs. Existing transformations into DSA perform a complex data flow analysis with exponential analysis time and work only for a limited set of input programs. Our method removes irregularities from the data flow by adding copy assignments to the program, and then it can use simple data flow analyses. The DSA transformation presented scales very well with growing program sizes and overcomes a number of important limitations of existing methods. We have implemented the method and it is being used in the context of memory optimization and verification of those optimizations.

1 Introduction

In a program in dynamic single assignment (DSA) form, there is only one assignment *at run time* to each scalar variable or array element[7]. Essentially this form directly encodes all the information gathered by an array data flow analysis, much like static single assignment (SSA) directly encodes use-def chains in the program[6]. While SSA focuses on scalar variables, DSA is geared towards array variables that are analyzed element by element. An alternate form of DSA is a system of recurrence equations[9] (SRE). This form does not give an execution order to the different statements, and instead only specifies the set of iteration points. SREs are usually used when the execution order of the original code does not matter, for example because a new execution order is determined anyway.

DSA form has essentially two advantages: analyses become simpler as each use can be linked to a single def by matching the array elements accessed, and there are more opportunities for reordering transformations. DSA form (and equivalently an SRE) is therefore used in a number of compiler techniques:

[*] Research supported by FWO Vlaanderen
[**] Supported by a specialization grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT)

1. **Parallelization.** The need for decreasing the restrictions on reordering transformations is necessary to get good results from parallelization[12,7]. SREs are often used as a starting point as it reveals all inherent parallelism in a program.
2. **Systolic arrays.** Procedural programs are transformed to SREs, which are mapped onto systolic arrays, forming the basis for VLSI chip design[16].
3. **Memory optimization.** Optimization of data-intensive programs, like multimedia and network applications, benefit from the use of DSA to simplify the transformations and to get a better result [15,5].
4. **Verification of source code transformations.** Verification [19,21] is shown to gain both simplicity and power by the use of DSA.

The contribution of this paper is a new method for transformation to DSA that offers two advantages over existing work:

1. It overcomes a number of limitations in existing methods. Our method is not limited to (possibly piecewise) affine expressions for loop bounds, conditionals and array indexation as are [7,10], but instead deals with general expressions including data-dependent expressions which are prevalent even in simple programs like Gaussian elimination with pivoting. The sole restriction is that constant bounds must be found on the loop iterators.
2. It is quadratic in the program size and polynomial in the depth of the loop nests, while existing methods are exponential.

Moreover the implementation has been successfully used in the context of memory optimizations [5] and verification of those transformations [19,21]. For more detailed results on the use with verification, see [20].

Section 2 discusses related work and situates our contribution. Section 3 introduces our approach. We explain the preparatory steps in Section 4, we do the DSA transformation in Section 5 and Section 6 discusses some additional steps. Section 7 rounds off the complexity analysis and Section 8 presents results of applying our prototype implementation to a number of multimedia kernels. Finally, Section 9 concludes.

2 Related Work

In optimizing compilers, data flow analyses are used to determine what optimizations are possible and how they should be done [1]. Forward data flow analyses propagate information from definitions of variables to their uses; they need to know which definitions reach which uses. One can distinguish the following queries (illustrated on the program in Fig. 1):

- **What definitions reach a given use?** The definitions `S1` and `S2` reach the use of `c` in statement `S2`.
- **Under what condition does each definition reach a given use?** For `S2` there are two reaching definitions. The reader can verify that `S1` reaches `S2` when `i` is 0 or when `j` is 6. `S2` reaches itself in all other cases.

```
for (k = 0; k <= 10; k++)
 c[k] = 0; //S1
for (i = 0; i <= 4; i++)
 for (j = 0; j <= 6; j++)
  c[i+j] = c[i+j]+a[i]*b[j]; //S2
```

Fig. 1. Polynomial multiplication 1

```
for (k = 0; k <= 10; k++)
 c[k] = k; //S1
for (i = 0; i <= 4; i++)
 for (j = 0; j <= 6; j++)
  c[i+j] = c[i+j]+a[i]*b[j]; //S2
```

Fig. 2. Polynomial multiplication 2

```
for (k = 0; k <= 10; k++)
 c1[k] = k; //S1
for (i = 0; i <= 4; i++)
 for (j = 0; j <= 6; j++)
  c2[i][j] = ( i==0 || j==6 ? c1[i+j] : c2[i-1][j+1] )+a[i]*b[j]; //S2
for (k = 0; k <= 10; k++)
 output(k <= 4 ? c2[k][0] : c2[4][k-4]); //S3
```

Fig. 3. Dynamic single assignment version of Fig. 2

– **What instance of a definition reaches what instance of a use?** Statement `S2` has an instance for each combination of values for `i` and `j`, and the reaching definition varies with the instance of `S2`. The instance for `i`$= i_1$ and `j`$= j_1$ is denoted `S2`(i_1, j_1). For `S2`(i_1, j_1) the reaching definition is `S1` when $i_1 = 0$ or $j_1 = 6$. The instance of `S1` that reaches `S2`(i_1, j_1) is for `k`$= i_1 + j_1$, denoted as `S1`$(i_1 + j_1)$.

The amount of information these three questions ask for increases, and so does the difficulty to answer them accurately. Classic compiler theory considers only the first question; this limits the amount of optimization, *e.g.,* in Fig. 1, `S1` assigns a constant 0 to `c`, but constant propagation would not be done because `c` changes value in the second loop. The information asked for in the first question can be encoded efficiently in the program (possibly in the intermediate representation) by putting it in static single assignment (SSA) form [6]. In SSA form there is *only one assignment to each variable in the program text*, making the search for reaching definitions an exercise in variable name matching. A ϕ-function is used to select the right reaching definition in case there is more than one.

The answer to the second question – if it can be answered for a particular case – can be regarded as the ϕ-functions of SSA form that are made explicit. So in Fig. 1, instead of assuming that the `c[i+j]` we read can be produced by either `S1` or `S2`, we can deduce that it is produced by `S1` when `i` is 0 or `j` is 6. This knowledge allows us to split `S2` on that condition and to do constant propagation to the part where the condition is true. The same technique is used to do copy propagation in [11], although it is not clear for which programs they can do this and how far they can go. Because it is typically useful for optimizing array-intensive programs – where the savings in memory accesses usually outweigh the cost of splitting statements – the form that encodes the answer to the second question explicitly could be referred to as array SSA (ASSA).

Finally, the answer to the third question completes the information about reaching definitions by indicating the exact instance of a statement that reaches a given use, which is done by giving the values of the iterators of surrounding loops. This information is necessary for optimizations such as general expression propagation. Consider Fig. 2 where the initialization of `c` to zero is replaced by an expression depending on the iterator `k`. If we want to propagate `S1` to `S2` again, we need to determine the correct value of `k` which will obviously depend on `i` and `j`. The third question supplies us with exactly this information. This information can be represented explicitly by putting the program in dynamic single assignment (DSA) form as presented by Feautrier in [7].

A program in DSA form assigns each array element or scalar variable only once during execution. The DSA form of the program in Fig. 2 is shown in Fig. 3. We have added an extra statement at the end to indicate where the final coefficients of the product end up being stored. For the conditional split-up we use the C operator `?:`. In statement `S2`, the condition selects between two reaching statements, while in statement `S3`, it selects between different expressions for the reaching instance. It is surprising how clear the answer to the questions above is from this program. For example `S2` reads both `c1` and `c2`, thus both `S1` and `S2` reach it. It reads `c1` when `i` is 0 or `j` is 6, hence that is when `S1` reaches `S2`. The exact element read by a given instance `S2`(i_2, j_2) is `c1[`$i_2 + j_2$`]`, so it is instance `S1`$(i_2 + j_2)$ that reaches *i.e.,* for `k` equal to $(i_2 + j_2)$. Despite the information being quite detailed, retrieving it is again simply a matter of matching.

Besides providing a compiler with very detailed data flow information, DSA form *can enable more transformations.* In DSA form, every value produced by an assignment is given its own memory location. After transformation to DSA we only have true dependencies which specify that the program should only read from a memory location after a value was written to it. As the name indicates, this kind of dependency is inherent to the program. DSA enables all reorderings of instructions as allowed by the true dependencies.

In [7] an automated method is proposed to translate code consisting of assignments to arrays that are arbitrarily nested in `for`-loops to DSA form, with the limitation that all loop bounds, conditions and indexation are affine expressions of loop iterators and a number of parameters. These limitations are relaxed slightly in [10] by allowing certain kinds of modulo and integer division in expressions for loop bounds, indexation and conditions. Methods to translate to SREs, *e.g.,* [4], lack the generality of [7] and [10]. However the approach of [7] can be adapted very easily to generate SREs, which is done by [2].

Finally we note that besides the simple categorization of SSA - ASSA - DSA, there are other variations in between such as [3] and [13].

3 Our Approach

Languages that impose DSA form, like Silage [8] or Single Assignment C [17], exist but are not widely used because DSA form can be awkward to write.

Most applications are written in a multiple assignment[1] form. Transformation to DSA form is tedious and error-prone when done manually, hence automation is needed.

The methods in [7,10] have limited applicability as they do not deal with programs that contain data dependent indexation or conditions, as is the case in *e.g.,* Gaussian elimination or motion estimation. This is in our experience an important limitation. Another disadvantage of [7] is that it breaks down for larger programs. The limited scalability is a problem since global optimizations inline functions to optimize over function boundaries. Indeed, aggressive optimizations usually require specializing each call-site anyway to obtain the best results.

Our method realizes a *scalable DSA transformation.* In practice it is linear in the program size (for constant loop depth). Our method *extends on existing methods*; it handles all pointer-free code with static memory allocation. It is restricted to a single function though, but this is not a real limitation because functions are inlined as argued above.

The most important obstacle for obtaining a scalable, generally applicable DSA transformation is that the three questions outlined in Sect. 2 need to be answered *exactly.* The data flow analysis presented in [7] gives exact answers, but because of that it is not generally applicable and scales badly – both in execution time and memory use. Data flow analyses as presented in [1] give approximate answers when control flow paths join, or when an array is accessed, and because of that are generally applicable and fast. The observation underlying our approach is that we can do away with approximations without sacrificing speed when *all variables are scalars, there are no* `if`*-statements, and every assignment is executed at least once in every iteration of every surrounding loop.* In case the program does not have these properties, we apply simple transformations, basically adding copy statements, until it does.

The fact that our DSA transformation method adds many copy operations to the transformed code seems to imply that it is useless in the context of optimization – it both increases memory use and increases the number of memory accesses. This is however no problem for functional verification of global transformations [19,21] which is very powerful but requires that programs be specified in DSA form. One of the transformations it is able to verify requires it to handle addition of copy operations, such that it can look right through the copy operations our method adds. This verification method can handle certain classes of data dependent conditions, but up to now there was no general way of transforming programs with data dependent conditions to DSA form. The presented work remedies that.

The DSA form we obtain can be made more usable by applying advanced copy propagation [22] on it. This removes the bulk of the added copy operations. Currently, advanced copy propagation deals with the same class of programs as the DSA method of [7]; we are extending it to handle the broader class of programs that our DSA method can transform.

[1] As opposed to (dynamic) single assignment.

```
for (t = 0; t < 1009; t++) {
  c[t] = 0; //S1
  for (i = 0; i <= t; i++)
    if (i > t - 1000 && i < 10)
      c[t] = c[t]+a[i]*b[t-i]; //S2
}
for (t = 0; t < 1009; t++)
  output_c(c[t]); //S3
```

(a) Our running example

n	maximum loop nest depth	2
d	maximum dimension of an array	1
a	number of assignments	2
l	number of loops	3
r	number of variable references	4
s	number of statements	7
v	number of variables	1
c	maximum number of conditions	6

(b) Program characteristics

Fig. 4.

Our running example is discrete convolution (Fig. 4(a)). The code assumes two signals are given. The first signal consists of 10 samples that are stored in array `a`. The second signal consists of `1000` samples that are stored in array `b`. The resulting 1009 samples are stored in array `c`. We focus on array `c` only from now on. Features of programs used in the complexity analysis along with the symbols used to denote them are shown in Fig. 4(b), whose right hand side column contains the corresponding values for our example program. Since we focus on `c`, the values shown in Fig. 4(b) ignore arrays `a` and `b`.

The DSA transformation consists of several steps that are described in subsequent sections. The general overview of the DSA transformation is shown in Fig. 5, with references to the corresponding sections.

4 Preparatory Steps

From array variables to scalars. The main difference between scalars and arrays is that scalars are assigned as a whole while arrays are typically assigned element per element. One way to handle this, as suggested in [6], is to introduce an `Update` operation for each assignment to an array element that builds a whole new array with all elements equal to the corresponding elements of the old array except for one element that gets a new value. The result for our running example is shown in Fig. 6(a). The `Ac` functions are technically not necessary as they can be trivially replaced by direct array references. The meaning of an `Up` function is shown in Fig. 6(b).

Every `Up` function has the old array as parameter (here it is still the same as the new array), a number of indexation expressions equal to the number of array dimensions, and the right hand side of the assignment. Setting up these functions is straightforward, and is $O(d)$ with d the maximum number of array dimensions in the program. For a assignments, this takes $O(d \cdot a)$ time.

Execute every assignment in every iteration. The two causes preventing the execution of an assignment are `if`-statements and loops that have zero iterations (possibly depending on the values of surrounding iterators). The simplest

```
transformToDSA(program)
  1. make arrays scalar (Section 4)
     for each right hand side array reference a :
       replace a with a Ac operation
     for each assignment a :
       replace the assignment by an Up operation
  2. pad with copy operations (Section 4)
     find constant loop bounds
     add no-op copy operations to the program

  3. do exact scalar data flow analysis (Section 5)
     determine for each variable reference what assignment
      instance wrote the value read, and under what condition
  4. transformation to DSA (Section 5)
     transform to DSA using the result of the data flow analysis

  5. expand Up and Ac (Section 6)
     replace each Up and Ac by its definition
```

Fig. 5. High-level script of DSA transformation

```
for (t = 0; t < 1009; t++) {
  c = Up(c, t, 0); //S1
  for (i = 0; i <= t; i++)
    if (i > t-1000 && i < 10)
      c = Up(c,t,Ac(c,t)+a[i]*b[t-i]); //S2
}
for (t = 0; t < 1009; t++)
  output_c(Ac(c, t)); //S3
```

(a) Fig. 4(a) with Up and Ac.

```
for (it = 0; it < 1009; it++)
  if (it == i)
    v1[it] = e;
  else
    v1[it] = v2[it];
```

(b) Meaning v_1 = Up(v_2,i,e).

Fig. 6. Fig. 4(a) with Up and Ac operations

way to overcome these problems is to transform away if statements and replace all loop bounds by constants. If these constant upper and lower bounds on iterator values indicate that the body of the loop is never executed, we remove it as the loop was useless to begin with.

For Fig. 6(a) the maximum value for the i iterator is 9. We can use this as upper bound on the i loop provided we move the condition that i should be smaller than t to the if-statement. The first step in removing the condition is to add an else-branch and add a no-op assignment for each variable assigned in the other branch. For our running example this results in Fig. 7(a). Note that now c is always assigned a value, regardless of the value of the condition of the if-statement. This can be made explicit in C as shown in Fig. 7(b). Now it is

```
for (t = 0; t < 1009; t++) {
  c = Up(c, t, 0); //S1
  for (i = 0; i < 10; i++)
    if (i > t-1000 && i <= t)
      c = Up(c,t,Ac(c,t)+a[i]*b[t-i]);
    else c = c; //S2
}
for (t = 0; t < 1009; t++)
  output_c(Ac(c, t)); //S3
```

(a) Padded with copy operations

```
P1
  for (t = 0; t < 1009; t++) {
P2
    c = Up(c, t, 0); //S1
  P3
    for (i = 0; i < 10; i++) {
  P4
      c = i > t-1000 && i <= t ?
        Up(c,t,Ac(c, t)+a[i]*b[t-i]):
        c; //S2
  P5
    }
P6
  }
P7
  for (t = 0; t < 1009; t++) {
P8
    output_c(Ac(c, t)); //S3
P9
  }
P10
```

(b) S2 assigns c in each iteration

Fig. 7. Removing conditions from Fig. 6(a)

trivial to see that when arriving at S3, the last assignment to c is done by S2 for t equal to 1008 and i equal to 9.

When all bounds and conditions are affine functions of the iterators, we can use linear programming [18] to find the extremal values for the iterators. Linear programming is polynomial in the problem size, which can be measured by the number of loop bounds and conditions governing a statement. If we call the maximum number of these conditions c, then finding two bounds for each of the l loops has a time complexity of $O(p(c) \cdot l)$ with $p(c)$ a polynomial in c. Adjusting the conditions guarding each assignment and adding a no-op assignment to each of these a assignments is a constant-time operation, thus giving an extra $O(a)$ for a total of $O(p(c) \cdot l + a)$.

5 Transformation to DSA Form

Reaching definitions analysis. In the following two steps, all array references will be changed to attain DSA form. First the left hand sides are transformed, then the right hand sides are changed accordingly using information about which definitions can reach each use. This information is obtained by running a reaching definitions analysis [1]. Because of our preparatory transformations, we can do this analysis exactly. Remember that we transformed all arrays to the equivalent of scalars, and that we made sure every assignment is executed at least once in each iteration of each surrounding loop.

The analysis can be performed for each variable separately because variables cannot interfere with the data flow of other variables. For our running example, the analysis for c is depicted graphically in Fig. 8. We define program points P_1 through P_{10}, one before and after each assignment, and one at the start and at end of each loop. At each program point we keep the reaching definition instances and the conditions under which they reach.

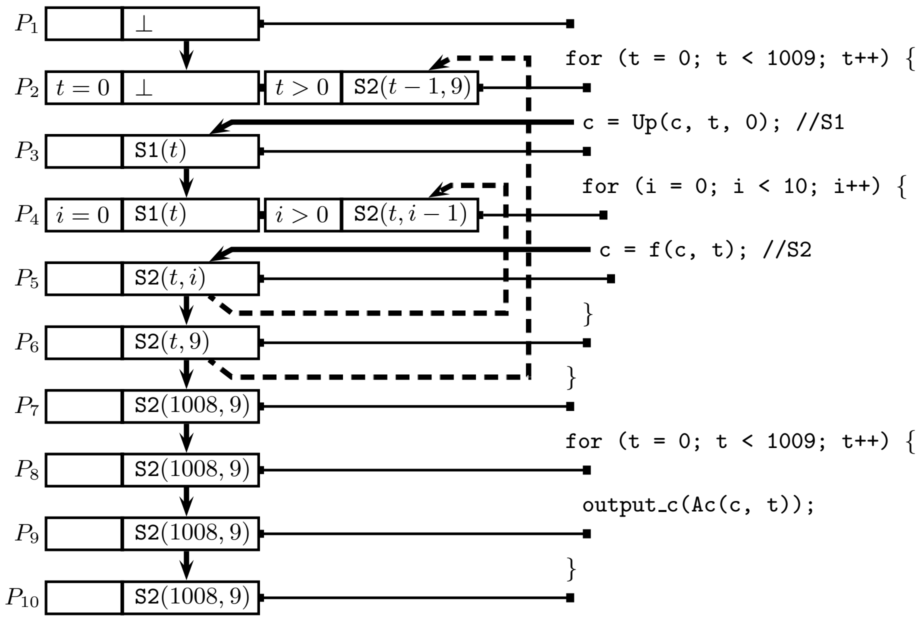


Fig. 8. Exact reaching definitions analysis on Fig. 7(b)

The number of reaching definitions to distinguish, and the conditions for each, are determined unambiguously by the following rules:

– At the start of the program (P_1), we distinguish one case and denote the reaching instance with $\perp$. This can be interpreted as either an uninitialized variable, or the initial value of the variable wherever it may come from.
– Upon entering a loop that contains an assignment to the variable we are analyzing (for example P_2), we split cases. Either we are in the first iteration of the loop ($\texttt{t} = 0$), in which case we distinguish the same reaching definitions as just before the loop, or we are in a later iteration ($\texttt{t} > 0$), in which case the assignment in the loop has been executed at least once and has killed all reaching definitions from before the loop. In the latter case the reaching definition is taken from the end of the previous iteration of the loop. In case of P_2, the end of the loop is P_6, and for $\texttt{t} > 0$, the reaching definition there is found to be S2$(t,9)$. Taking into account that we are in the next iteration of the t loop, the reaching definition to P_2 becomes S2$(t-1,9)$.
– Upon entering a loop that contains no assignment to the variable being analyzed (*e.g.,* P_8), the reaching definitions are obviously not affected and hence they are just copied from before the loop.
– Just after a loop (*e.g.,* P_6), the reaching definitions are those that reach the end of the last iteration of that loop. The reaching definitions to P_6 are those at P_5 for $\texttt{i} = 9$. There is only one, and filling in $\texttt{i} = 9$ gives S2$(t,9)$.

```
for (t = 0; t < 1009; t++) {
 if (t == 0) c1[t] = Up(c, t, 0); //S1
 else c1[t] = Up(c2[t-1][9], t, 0); //S1
 for (i = 0; i < 10; i++)
   if (i == 0) c2[t][i] = i > t-1000 && i <= t ?
                Up(c1[t],t,Ac(c1[t],t)+a[i]*b[t-i]) : c1[t]; //S2
   else c2[t][i] = i > t-1000 && i <= t ?
          Up(c2[t][i-1],t,Ac(c2[t][i-1],t)+a[i]*b[t-i]) : c2[t][i-1]; //S2
}
for (t = 0; t < 1009; t++)
 output_c(Ac(c2[1008][9], t)); //S3
```

Fig. 9. Converting Fig. 7(b) to full DSA form

– Directly after an assignment (*e.g.,* P_5) all other reaching definitions are killed. Thus the assignment is the only reaching definition, and the reaching instance is obviously the one from the current iteration of the loop. For P_5 this is S2(t, i).

The maximum number of reaching definitions to discern at each program point is one more than the loop depth. At the start of the program we start with one case and zero loop depth. When going through a loop without an assignment of interest, nothing changes and the number of cases is smaller than the maximum. When entering a loop with an assignment of interest, one case is added (hence the maximum number of cases is at worst one more than the loop depth). After the assignment, the number of reaching definitions are reset to one, which is also the number of reaching definitions at the end of the loop. With n the maximum loop depth, the amount of information kept at each program point is $O(n^2)$. Calculation of this information for all program points, $O(s)$ in number, and for all variables, v in number, requires a total time $O(s \cdot n^2 \cdot v)$. This is because all information can be calculated without any complex bookkeeping by just going over the program twice.

Transformation to DSA form. In multiple assignment code involving only scalars, there are two causes of multiple assignments to the same variable; either there are two assignments to the same variable, or an assignment is in a loop. The former cause is removed by renaming the variables in the left hand side of each assignment such that they each write to a different variable. Typically this is done by adding a sequence number at the end of the variable. The latter cause is removed by changing the variable assigned to an array with one dimension for each surrounding loop, and we index that dimension with the iterator of the corresponding loop. For our running example this is shown in Fig. 9.

After adjusting the left hand sides, we still need to adjust the variable accesses in the right hand sides. It is possible that a number of assignments (definitions) reach a variable access, and which one needs to be read from depends on the values of the iterators. This information was derived by applying the reaching definitions analysis previously described.

For our example, $\texttt{S1}(t)$ is reached by $\perp$ when $t = 0$. In other words when `S1` is executed for `t` equal to 0, we should read from the variable written by the implicit assignment at the start of the program, which is just `c`. If $t > 0$, $\texttt{S1}(t)$ is reached by $\texttt{S2}(t-1, 9)$. So we should read from the variable written by $\texttt{S2}(\texttt{t} - 1, 9)$, which is `c2[t-1][9]`. Because of the special form of the left hand sides, the correspondence between $\texttt{S2}(t-1, 9)$ and `c2[t-1][9]` is direct. Filling everything in in Fig. 7(b) results in Fig. 9.

Adding the sequence number to the left hand side of assignment is a constant time operation, and because we need to add a number of dimensions equal to the number of surrounding `for`-loops, this step is $O(n \cdot a)$ with n the maximum loop nest depth for each of the a assignments. The adjustment of the right hand sides is executed for each variable reference, r in number. This requires little more than just copying the expressions from the calculated reaching definitions, whose worst case size for each reference is $O(n^2)$. The complexity of this step is thus $O(r \cdot n^2 + n \cdot a)$.

6 Additional Steps

Expansion of `Up` and `Ac`. Finally, we need to replace the `Up` and `Ac` function by their respective implementations. This is straightforward given the simple definition of these two functions.

There is an `Up` function for every assignment – at most a in number – which on expansion gets an extra number of loops – one for each dimension of the original array such that the whole array can be written. The maximal dimension of the array reference is then $n + d$, and thus the size of the generated array references is $O(n + d)$. Since the generation is direct, the expansion of all `Up` functions requires a time that is $O((n + d) \cdot a)$. There is an `Ac` function at every array reference in the original program – at most r in number – which should be replaced with a new array reference with maximum dimension again $n + d$. The expansion of all `Ac` functions is then $O((n + d) \cdot r)$. The total time needed for this step is $O((n + d) \cdot (a + r))$.

Extensions. We have described our DSA transformation for the case where all loop bounds, indexation and conditions are affine functions of surrounding iterators. This fact is only used in Sec. 4 to find a constant upper and lower bound for the iterators. Subsequent steps work with those constants instead of the original loop bounds. The sole preceding step is the introduction of the `Up` and `Ac` functions, but they take along the indexation literally without analyzing it. This is illustrated in Fig. 6(b) where *index* is copied literally.

In general, conditions can be data dependent or non-affine, which prevents us from simply using linear programming to find the upper and lower bounds for the loops. We distinguish between bounds containing modulo operations and general non-affine bounds. When the bounds do contain modulo, we can translate the problem of finding a constant upper and lower bound to an equivalent problem that is affine. It is well known that modulo and integer division can be modeled using existential variables, *e.g.,* saying that $a = b \bmod c$ is the same as saying that $\exists q \in \mathbb{Z} : 0 \leq b - q \cdot c < c \wedge a = b - q \cdot c$. For the purpose of finding bounds, we can drop the requirement that q should be integer. The condition has become linear and q is just an extra variable in the linear programming problem.

For general non-affine expressions, including data dependent ones, we can take several approaches:

– Find the extremal values for the iterators in the iteration domain with all expressions that are not affine functions of the iterators discarded. This does not work when the only expression bounding an iterator is not affine, but it was applicable in most experiments we have performed.
– Use the range of the data type of the iterator as upper and lower bound. This is a crude approximation, but it does not affect our intended use. For both verification and optimizations like advanced copy propagation, the iteration domains would become larger measured by the number of points in it, but their representation stays the same size – only a constant needs to be changed.
– A more advanced technique analyzes the range of values that data can take and uses this to find a tighter bound for data dependent conditions. This analysis is actually easier on DSA programs and should be performed after the DSA transformation to tighten the bounds. This is outside the scope of this paper.

With the above extensions we can conclude that our approach works for general kinds of conditions, loop bounds and indexation.

7 Complexity Analysis

Since the whole DSA transformation method is the succession of each of the separate steps, the total complexity is simply the sum of the complexities of all steps. This results in a grand total of $O(d \cdot a + (p(c) \cdot l + a) + s \cdot n^2 \cdot v + n \cdot a + r \cdot n^2 + (n + d) \cdot (a + r)) = O(d \cdot (a + r) + n \cdot (s \cdot n \cdot v + r \cdot n + a) + p(c) \cdot l + a)$. Assuming that loop depth and array dimension are bounded by a constant, this simplifies to $O(r + a + s \cdot v + p(c) \cdot l)$. Because every measure in this formula is worst case proportional to the size of the program, our method is polynomial in the size of the program. c is typically proportional to the depth of nesting, hence we can reasonably assume that $p(c)$ is bounded by a constant as well, and we obtain $O(a + s \cdot v + r + l)$. The presence of $s \cdot v$ in this expression is due to the reaching definitions analysis, and this is the only term that could have a tendency to grow large because as the number of statements s increases, the number of variables v is likely to increase as well. This makes our method quadratic. However, our experiments suggest that our method tends towards linearity. The difference between our experiments and our analysis is probably caused by the fact that the big-O notation discards the constant coefficients

Table 1. Benchmarks with transformation times

benchmark	LOC	n	time	Feaut.
*gauss*1	16	2	0.024s	X
*gauss*2	17	2	0.027s	X
*gauss_dd*1	65	3	0.331s	
*gauss_dd*2	64	3	0.284s	
*mp*3_1	75	4	4.946s	
*mp*3_2	66	4	1.898s	
qsdpcm	495	12	41.950s	
*cavdet*1	70	4	1.275s	
*cavdet*2	96	4	0.820s	
*cavdet*3	93	4	0.762s	
*cavdet*4	53	4	0.425s	
*cavdet*5	53	4	0.408s	
*cavdet*6	54	4	0.388s	
*cavdet*7	54	4	0.388s	
*cavdet*8	53	4	0.310s	
*cavdet*9	61	4	0.651s	

(when some complexity is $O(2 \cdot n)$, it is also $O(n)$ and vice versa). This seems to indicate that the coefficients of the linear parts of the complexity weigh heaviest – most notably $p(c)$ can be quite a large constant.

8 Experimental Results

We have implemented our DSA transformation using the Omega library [14] for modeling and simplifying the iteration domains and for finding the extremal values of the iterators. Our implementation has full support for data dependent indexation and conditions, and partial support for extra data types from C (*e.g.,* `struct`s).

We have applied the transformation to a number of multimedia kernels: Gaussian elimination without pivoting (gauss1 and gauss2) and with pivoting (*gauss_dd*1 and *gauss_dd*2), a cavity detector (*cavdet*1 through *cavdet*9), a MP3 decoder (*mp*3_1 and *mp*3_2) and QSDPCM motion estimation (*qsdpcm*). These benchmarks are shown in Table 1. The LOC column lists the number of lines of code, and the n column lists the maximum loop nest depth. The DSA transformation was run on a Pentium 4 2.4GHz with 768 MB RAM.

Of these benchmarks, only gauss1 and gauss2 do not contain data dependent indexation or conditions and can be handled by [7]. For example *gauss_dd*1 and *gauss_dd*2 search for a pivot and then swap rows depending on the location of the pivot. This location depends on the original matrix elements in a non-trivial way and changes with each iteration. This makes that these programs cannot easily be transformed to a form that [7] would be able to handle. The benchmarks that can be handled by [7] are marked in the last column of Table 1.

For each benchmark (except *qsdpcm*) the first version is the original version of the code. The other versions are obtained by applying transformations in order to increase locality of access as well as regularity of access usually by aligning loops and merging them. The effect is that uses and definitions of array elements are moved closer to each other. Because life times of array elements become much smaller, fewer elements are live at each point and less memory is needed to store them. An effect noticeable from Table 1 is that DSA transfor-

Table 2. Scalability experiment

k	1	2	3	4	5	6	7	8	9	10	25	50
$t(s)$	1.362	2.679	3.986	5.303	6.612	7.940	9.254	10.56	11.89	13.20	32.93	65.95
t/k	1.362	1.339	1.329	1.326	1.322	1.323	1.322	1.320	1.321	1.320	1.317	1.319

mation times tend to go down with the amount of locality/regularity improving transformations applied. This is because when the reaching definitions are close to the uses, they are easier to find. Additionally, fewer reaching definitions need to be discerned and the conditions governing them become smaller. The beneficial effect of regularity and locality can even outweigh an increase in program size, as illustrated by *cavdet*1 vs *cavdet*2. Note that the time for *cavdet*9 is longer than for *cavdet*8, which seems to go against the general trend. But in *cavdet*9 heavily used array elements are copied into smaller arrays, which can be placed in a smaller memory that is faster and consumes less energy [5]. Besides making the program larger, this makes the data flow more complex again. Intuitively, the reason is that these arrays are added when there are many uses for a single definition, and this single definition usually cannot be put close to each use.

To ascertain that our method scales well with growing program size, we have run our DSA transformation on a program of growing size. Usually two programs of differing size have a different loop structure as well, even if the maximum loop depth is the same. This causes a lot of noise on the measurements. A very clear example is the difference between *gauss*1 and *qsdpcm*: an increase of roughly a factor 30 in LOC, but an increase of 1750 in transformation time. At play here is the great increase in loop depth. To reduce this noise we have opted to chain the cavity detector kernel (an adaptation of *cavdet*1) together a number of times (denoted k) with the output array of one instance of the cavity detector the same as the input array of the next instance. This way we can build reasonable programs of a size that is a multiple of the original cavity detector, but with the extra that the code that is added each time has a comparable loop structure. The measurements are shown in Table 2.

The transformation time is shown as t, and the transformation time per instance of the cavity detector is shown as the ratio t/k. This ratio actually decreases, but this is due to a constant startup cost of which each instance's share decreases as the number of instances increases. Another measure for the transformation time for one instance of the cavity detector could be the difference in value for t for $n = 1$ and $n = 2$. This difference is 1.317 which t/k seems to converge to, increasing our confidence that our method indeed behaves linearly.

For comparison with the DSA transformation of [7], we have done the scalability experiment with MatParser [10] as well. MatParser is a mature, heavily optimized tool that is believed ready for commercial use. This makes it a good candidate for comparison. Because MatParser is not publicly available, the experiments with MatParser had to be done on a different machine: a Pentium 4 2.8GHz with 768 MB RAM. The results of this experiment are plotted in Fig. 10. The timings for MatParser only go up to $k = 5$ because for larger

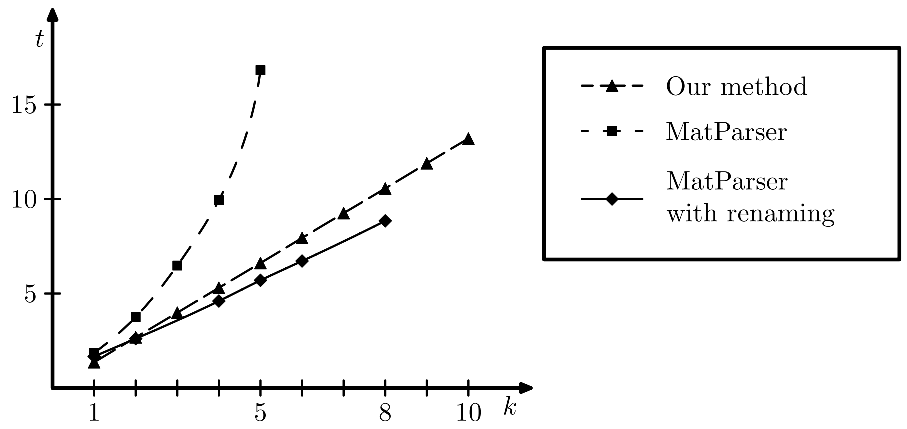


Fig. 10. Comparison with MatParser of the transformation time as function of the number of instances of the cavity detector

k MatParser runs out of memory. There are too few data points to derive whether the behavior is exponential, but it is in any case not linear. As it turns out the problem with MatParser (in this case, but also in general) is that it tends to handle many assignments to the same variable badly. This is a problem specifically highlighted by our scalability experiment as each assignment to a variable is repeated k times.

Repeating the same experiment, but with each instance of the cavity detector given its own private set of variables (by renaming them), should show improvement in MatParser's performance. The results are shown in Fig. 10 as well. The first thing to notice is that now it is possible to get up to $k = 8$ before running out of memory. Second thing to notice is that the data points lie on a straight line now, indicating that many assignments to the same variable do indeed pose a problem for MatParser. An important thing to notice is that for our tool the experiment with variable renaming results in exactly the same graph as without variable renaming. This is an important feature of our method: it shows a remarkable capability to discern different uses of the same variable, which in this case are the uses of a variable by each instance of the cavity detector. Another experiment with the *qsdpcm* benchmark shows that MatParser has problems with deeply nested loops for the same reason. Because MatParser considers the dependence over each surrounding loop separately, the statements surrounded by the 12-dimensional loop are essentially split in 12 separate statements, causing MatParser to go hopelessly out of memory.

9 Conclusion

In this paper we have presented a new method that transforms programs to dynamic single assignment which is in practice linear in the size of the program (for a constant loop depth). This is achieved by adding copy operations in such a way that we can use an exact scalar reaching definitions analysis and a simple way of determining the indexation for each of these reaching definitions whereas existing methods need an expensive, exact array data flow analysis. We have implemented this DSA transformation and it is currently being used as an enabling step for functional verification [19,21] and memory optimizations [5]. The extra copy operations can be removed by advanced copy propagation [22]. We plan to investigate the interaction between our new DSA transformation and copy propagation as part of future work. On the one hand experiments show that for non-data-dependent code (*e.g., gauss*1 and *gauss*2 from Fig. 1) all copy operations can be removed, and the resulting code is equivalent to the code Feautrier's method returns. There is the option though to leave some of the copy operations. This would allow to control the complexity of the transformation as well as the code complexity of the resulting program, as both are tightly linked. This complexity is due to the number of cases that need to be discerned, often due to border conditions which give rise to few copy operation instances that are not worth removing given the cost of doing so. The investigation of the trade-off of remaining overhead versus transformation time and code complexity is left for future work. On the other hand, for programs involving data-dependent indexation or conditions, not all copy operations can be removed as that would imply that the program is fully analyzable which is not so in general. In future work, we want to determine how far copy propagation can go in general.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Inc., 1986.
2. C. Alias. `f2sare`. http://www.prism.uvsq.fr/users/ca/progs/f2sare.tgz, 2003.
3. R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The program dependence web: A representation supporting control-, and demand-driven interpretation of imperative languages. In *Proceedings of PLDI'90*, pages 257–271, 1990.
4. J. Bu and E. Deprettere. Converting sequential interative algorithms to recurrent equations for automatic design of systolic arrays. In *Proceedings of ICASSP '88*, volume vol IV, pages 2025–2028. IEEE Press, 1988.
5. F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design.* Kluwer Academic Publishers, 1998.
6. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
7. P. Feautrier. Array expansion. In *Proceedings of the Second International Conference on Supercomputing*, pages 429–441, St. Malo, France, 1988.
8. D. Genin and P. Hilfinger. *Silage Reference Manual, Draft 1.0.* Silvar-Lisco, Leuven, 1989.
9. R. Karp, R. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14:563–590, July 1967.

10. B. Kienhuis. Matparser: An array dataflow analysis compiler. Technical report, University of California, Berkeley, February 2000.
11. K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *Symposium on Principles of Programming Languages*, pages 107–120, 1998.
12. Z. Li. Array privatization for parallel execution of loops. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 313–322, New York, NY, USA, 1992. ACM Press.
13. C. Offner and K. Knobe. Weak dynamic single assignment form. Technical Report HPL-2003-169, HP Labs, Aug 2003.
14. W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing '91*, Albuquerque, NM, 1991.
15. F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages*, 22(5):773–815, 2000.
16. P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *ISCA '84: Proceedings of the 11th annual International Symposium on Computer Architecture*, pages 208–214, New York, NY, USA, 1984. ACM Press.
17. S.-B. Scholz. Single Assignment C - Functional Programming Using Imperative style. In *Proceedings of the 6th International Workshop on Implementation of Functional Languages (IFL'94)*, pages 21.1–21.13, 1994.
18. A. Schrijver. *Theory of Integer and Linear Programming.* John Wiley and Sons, 1986.
19. K. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. An automatic verification technique for loop and data reuse transformations based on geometric modeling of programs. *Journal of Universal Computer Science*, 9(3):248–269, 2003.
20. K. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Automatic verification of source code transformations on array-intensive programs: demonstration with real-life examples. Technical Report CW 401, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2005. In preparation.
21. K. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Verification of source code transformations by program equivalence checking. In *Compiler Constrcution, 14th International Conference, CC 2005, Proceedings*, volume 3443 of *LNCS*, pages 221–236. Springer, 2005.
22. P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor. Advanced copy propagation for arrays. In *Languages, Compilers, and Tools for Embedded Systems LCTES'03*, pages 24–33, June 2003.

Abstract Dependences for Alarm Diagnosis

Xavier Rival

École Normale Supérieure,
45, rue d'Ulm,
75230, Paris cedex 5, France

Abstract. We propose a framework for dependence analyses, adapted –among others– to the understanding of static analyzers outputs. Static analyzers like ASTRÉE are sound but not complete; hence, they may yield false alarms, that is report not being able to prove part of the properties of interest. Helping the user in the alarm inspection task is a major challenge for current static analyzers. Semantic slicing, i.e. the computation of precise abstract invariants for a set of erroneous traces, provides a useful characterization of a possible error context. We propose to enhance semantic slicing with information about abstract dependences. Abstract dependences should be more informative than mere dependences: first, we propose to restrict to the dependences that can be observed in a slice; second, we define dependences among abstract properties, so as to isolate abnormal behaviors as source of errors. Last, stronger notions of slicing should allow to restrict slices to such dependences.

1 Introduction

In the last few years, many static analyzers were developed so as to answer the need for certification methods and to check that critical programs satisfy certain correctness properties, such as memory properties [21], the safety of pointer operations [25], the absence of buffer overruns [15], or the absence of runtime errors [11,5]. These tools should produce sound results (they should not claim any false property to hold) and be automatic (they infer program invariants for the certification instead of asking the user to provide the invariants and just check them). Due to the undecidability of the properties they intend to prove, these tools are necessarily incomplete: they may report *false alarms*, i.e. critical operations they are not able to prove safe. From the user point of view, an alarm could be either a true error, or a false alarm (which may be non-trivial to check manually); hence, alarm inspection is a major issue in static analysis.

In the case of ASTRÉE, a static analyzer for proving the absence of runtime-errors in large C programs, a lot of work was done in order to make the analyzer precise, i.e. reduce the number of false alarms [5]; this approach allowed us to reduce the number of false alarms to 0 in some families of programs. Then, our previous work [23] proposed *semantic slicing* as a way to approximate precisely a set of executions satisfying some conditions; it may help to prove an alarm false or to make the alarm diagnosis process easier. Among possible *criteria*

$X, X0, X1, X2, X3$	l_0 **while**(**true**){	l_5 $t[1] = X1;$
floating point variables	l_1 **input**($X0 \in [-100., 100.]$);	l_6 $X3 = X0 + X1;$
t, floating point array of length 2	l_2 **input**($X1 \in [-50\,000, 50\,000]$);	l_7 $X = X3 + X2;$
initializations:	l_3 $X2 = -0.5 * t[0] + 5 * t[1] + X;$	l_8 }
$t[2] = \{0, 0\}; X = 0;$	l_4 $t[0] = t[1];$	$l_9 \ldots$

Fig. 1. An unstable retroaction

for defining semantic slices, we can cite the data of a (set of) final state(s) (e.g. states which may lead to an error), of conditions on the inputs, and of "execution patterns" which specify sets of control flow paths (described, e.g. by automata). Semantic slices are helpful in the alarm inspection process. Yet, the amount of data to investigate may still be fairly important. Moreover, [23] requires the user to provide the semantic slicing criteria, so we wish to help the user with a more automatized process, even though these criteria are usually rather simple.

We propose to reinforce the basic dependence analysis implemented in [23] with more restrictive analyses, producing fewer dependences supposed to be more "related" to the alarm under investigation. More precisely, we intend to restrict to dependences that can be *observed* on a set of program executions corresponding to an alarm and to compute abstract dependences, i.e. chains of dependences among *abstract properties* likely to capture the cause for an alarm. Such dependences should help making the alarm inspection process more automatic, by providing good candidate slicing criteria. For instance, in the example of Fig. 1, an unstable retroaction causes $X, X2$ to diverge: in case the input $X1$ is large for all iterations, X will grow, and will eventually overflow –whatever $X0$. ASTRÉE discovers two alarms at l_3 and l_7. Semantic slicing allows to inspect sets of diverging traces but does not lead to the causes for the divergence. We expect some dependence analysis to provide some hint about what part of the program to look at; for instance, the input $X0$ plays little role in the alarm compared to $X1$, so we would expect to rule it out, which is not achieved by classical slicing [26], conditioned slicing [6], or semantic slicing [23] methods. Last, the cyclic dependence among "diverging" variables $(X, X2)$ should suggest to unroll the loop in order to study the divergence; this information may be used to inferring semantic slicing criteria automatically, thus enhancing [23].

The contribution of this paper is both theoretical and practical:

– we introduce alternative, more selective notions of dependences and propose algorithms for computing them; we also propose new notions of non-executable, but analyzable program slices;
– we illustrate these concepts with examples and case studies, together with early implementation results; moreover, we show how these dependences help for better semantic slicing and more efficient alarm inspection.

Sect. 2 defines observable and abstract dependences and shows the relevance of these notions. Sect. 3 provides an ordering among abstract dependences. Sect. 4 focuses on the approximation of observable dependences. Sect. 5 tackles the case of abstract dependences. Sect. 6 presents a few case studies. Sect. 7 concludes and reviews related work.

2 Dependences Framework

2.1 Basic Notations

We let $\mathbb{X}$ (resp. $\mathbb{V}$) denote the set of variables (resp. of values); we write $\mathbb{e}$ (resp. $\mathbb{s}$) for the set of expressions (resp. statements, aka programs). We assume that $\mathbb{X}$ is finite. A variable (resp. value) has scalar or boolean type. An expression is either a constant $v \in \mathbb{V}$, a variable $x \in \mathbb{X}$, or the application $e_0 \oplus e_1$ of a binary operator $\oplus$ to a pair of expressions $e_0, e_1 \in \mathbb{e}$. A statement is either an assignment $x = e$ (where $x \in \mathbb{X}$, $e \in \mathbb{e}$), a conditional $\mathbf{if}(e)\{s_0\}\mathbf{else}\{s_1\}$ (where $e \in \mathbb{e}, s_0, s_1 \in \mathbb{s}$), a loop $\mathbf{while}(e)\{s_0\}$, a sequence of statements $s_0; \ldots; s_1$, or an $\mathbf{input}(x \in V)$ statement which writes a random value chosen in $V \subseteq \mathbb{V}$ into variable x. We do not consider more involved C data and control structures (pointers, unions, functions, recursion) so as to make the presentation less technical. The control point before each statement and at the end of each block is associated to a unique label $l \in \mathbb{L}$.

We let $\mathbb{S}$ denote the set of states; a state is defined by a control state $l \in \mathbb{L}$ and a memory state $\rho \in \mathbb{M}$, so that $\mathbb{S} = \mathbb{L} \times \mathbb{M}$. An execution (or *trace*) σ of a program is a finite sequence of states $\langle (l_0, \rho_0), \ldots, (l_n, \rho_n) \rangle$ such that $\forall i,\ (l_i, \rho_i) \rightarrow (l_{i+1}, \rho_{i+1})$ where $(\rightarrow) \subseteq \mathbb{S}^2$ is the transition relation of the program; Σ is the set of all traces. For instance, in the case of the assignment $l_0 : x := e; l_1$, there is a transition $(l_0, \rho) \rightarrow (l_1, \rho[x \leftarrow [\![e]\!](\rho)])$, where $[\![e]\!] \in \mathbb{M} \rightarrow \mathbb{V}$; in the case of the input statement $l_0 : \mathbf{input}(x \in V); l_1$, then $(l_0, \rho) \rightarrow (l_1, \rho[x \leftarrow v])$, where $v \in V$. The semantics $[\![s]\!]$ of program s collects all such traces. If P is a set of stores and $x \in \mathbb{X}$, we write $P(x)$ for $\{\rho(x) \mid \rho \in P\}$.

2.2 Dependences on Functions

Our purpose is to track the following kind of dependences: we would like to know what observation of the program (that is, which variable, and at which point) may affect the value (or some abstraction of it) of variable x at point l. We give a definition for "classical" dependences in the case of functions first and extend this definition to the dependences in control flow graphs afterwards; extended definitions are provided in the next subsections. We write $\mathfrak{Den}$ for $\mathbb{M} \rightarrow \mathcal{P}(\mathbb{M})$.

Definition 1 (Classical dependences). *Let $\phi \in \mathfrak{Den}$, $x_0, x_1 \in \mathbb{X}$. We say that ϕ induces a* dependence *of x_1 on x_0 if and only if there exist $\rho_0 \in \mathbb{M}$, $v_a, v_b \in \mathbb{V}$ such that $\phi(\rho_a)(x_1) \neq \phi(\rho_b)(x_1)$ where $\rho_i = \rho_0[x_0 \leftarrow v_i]$. Such a dependence is written $x_1 \overset{\phi}{\rightsquigarrow} x_0$ (or $x_1 \rightsquigarrow x_0$ when there is no ambiguity about the function ϕ). Last, we let $\mathfrak{D}_{\mathrm{f}}[\phi]$ denote the set of dependences induced by ϕ: $\mathfrak{D}_{\mathrm{f}}[\phi] = \{(x_0, x_1) \mid x_1 \overset{\phi}{\rightsquigarrow} x_0\} \in \mathfrak{Dep}_{\mathrm{f}}$ (we write $\mathfrak{Dep}_{\mathrm{f}} = \mathcal{P}(\mathbb{X}^2)$).*

Intuitively, there is a dependence of x_1 on x_0 if a single modification of the input value of x_0 may result in a different result for x_1. This definition is comparable to the notion of non-interference [18]; in fact the occurrence of a dependence corresponds to the opposite of non-interference. It can also be related to the notions of secure information flow [13]. Our motivation is to investigate the origin of some (abnormal) results, which is clearly related to the information flows to the alarm location. Usual notions of slicing require more dependences to be taken into account, since they aim at collecting all parts of the program that play a role in the computation of the result (constant parts are required even if they do not affect the result; their case will be considered in Sect. 6).

Example 1 (Dependences of functions). Let $x, y \in \mathbb{X}$. Let us consider the function $\phi \in \mathfrak{Den}$ defined by $\phi(\rho) = \{\rho[y \leftarrow \rho(x)]\}$ if $\rho(b) = \mathbf{true}$ and $\phi(\rho) = \emptyset$ if $\rho(b) = \mathbf{false}$. Then, if $\rho_0 \in \mathbb{M}$, and $z \in \mathbb{X}$, $\phi(\rho_0[b \leftarrow \mathbf{false}])(z) = \emptyset \neq \phi(\rho_0[b \leftarrow \mathbf{true}])(z)$; hence, $z \overset{\phi}{\rightsquigarrow} b$. Similarly, we would show that $y \rightsquigarrow x$. Last, if $z \in \mathbb{X} \backslash \{y\}$, we could prove that $z \rightsquigarrow z$, and that ϕ has no other dependence.

The definition of dependences among variables in a control flow graph derives from the above definition and the classical abstraction of sets of traces into functions [9]. Indeed, let $l_0, l_1 \in \mathbb{L}$, $x_0, x_1 \in \mathbb{X}$. Then, we shall approximate the set of traces starting at l_0 and ending at l_1 with a function in $\mathfrak{f}_{l_0}^{l_1} \in \mathfrak{Den}$ defined by $\mathfrak{f}_{l_0}^{l_1}(\rho_0) = \{\rho_1 \mid \exists \langle (l_0, \rho_0), \ldots, (l_1, \rho_1) \rangle \in [\![s]\!]\}$. We say that s induces a dependence of (l_1, x_1) on (l_0, x_0) if and only if $(x_0, x_1) \in \mathfrak{D}_{\mathrm{f}}[\mathfrak{f}_{l_0}^{l_1}]$ (such a dependence will be denoted by $(l_1, x_1) \rightsquigarrow (l_0, x_0)$). Last, we note $\mathfrak{D}_{\mathrm{t}}[s]$ for $\{((l_0, x_0), (l_1, x_1)) \mid l_0, l_1 \in \mathbb{L},\ (x_0, x_1) \in \mathfrak{D}_{\mathrm{f}}[\mathfrak{f}_{l_0}^{l_1}]\}$ (we write $\mathfrak{Dep}_{\mathrm{t}}$ for $\mathcal{P}((\mathbb{L} \times \mathbb{X})^2)$).

Example 2 (Ex. 1 continued). Let us consider the program fragment $l_0 : \mathbf{if}(b)\{l_1 : y = x; l_2 : \ldots\} \ldots$. Then, the function $\mathfrak{f}_{l_0}^{l_2}$ corresponds to the function ϕ introduced in Ex. 1. Therefore, the set of dependences between l_0 and l_2 is $\mathfrak{D}_{\mathrm{f}}[\mathfrak{f}_{l_0}^{l_2}] = \{(b, y); (x, y)\} \cup \{(v, z) \mid z \in \mathbb{X} \setminus \{y\},\ v = z \vee v = b\}$.

In the following, we rely on this straightforward extension of the definition of dependences induced by functions into dependences induced by programs (so that we do not have to state it again). It is important to note that $\mathfrak{D}_{\mathrm{f}}[]$ is *not* monotone; in particular the greatest element of $\mathfrak{Den}$ is $(\lambda\rho.\mathbb{M})$ and induces *no* dependence. The purpose of the following subsections is to select *some* dependences that should be relevant to the problem under consideration.

2.3 Observable Dependences

We consider now the problem of restricting the dependences that can be observed on a subset of traces (aka a *semantic slice*). The semantic slice is usually defined by a criterion c chosen in some domain $\mathbb{C}$; moreover, we assume a concretization function $\gamma_{\mathbb{C}} : \mathbb{C} \to \mathcal{P}(\Sigma)$ describes the meaning of semantic slicing criteria in terms of sets of traces. For instance, we may fix sets of initial and final states; hence, $\mathbb{C} = \mathcal{P}(\mathbb{S}) \times \mathcal{P}(\mathbb{S})$ and $\gamma_{\mathbb{C}}(\mathcal{I}, \mathcal{F}) = \{\langle s_0, \ldots, s_n \rangle \in \Sigma \mid s_0 \in \mathcal{I} \wedge s_n \in \mathcal{F}\}$; in the end the semantic slice (traces starting in $\mathcal{I}$ and ending in $\mathcal{F}$) boils down to $[\![s]\!] \cap \gamma_{\mathbb{C}}(\mathcal{I}, \mathcal{F})$. Other useful examples of semantic slices were introduced in [23] (input constraints and restriction to some execution patterns).

We let $\mathcal{E} \subseteq \Sigma$ denote a semantic slice and $\mathcal{E}^\sharp : \mathbb{L} \to \mathcal{P}(\mathbb{M})$ denote an "abstraction" [10] for the semantic slice $\mathcal{E}$: if $\langle \ldots, (l, \rho), \ldots \rangle \in \mathcal{E}$, then $\rho \in \mathcal{E}^\sharp(l)$. In practice, $\mathcal{E}^\sharp$ is computed by a static analyzer like ASTRÉE [5] ($\mathcal{E}^\sharp(l)$ is the

```
                         l0  if(b){
                         l1     y = x;
Initial condition (l0):  l2  }else{
  b = true               l3     ...
                         l4  }
                         l5  ...
                   (a)
```

```
Initial condition (l0):  l0  if(x > 5){
  x ∈ [0, 10]            l1     y = 1 000 ⋆ x;
  y ∈ [0, 5]             l2  }else{
  z ∈ [−4, 15]           l3     y = y + z;
Final condition (l5):    l4  }
  y ≥ 1 000              l5  ...
                   (b)
```

Fig. 2. Observable dependences

concretization of the local, abstract numerical invariant at point l). Moreover, we assume that $\mathcal{E}$ satisfies a *closeness assumption*: $\langle s_0, \dots, s_n, \dots, s_m \rangle \in \mathcal{E} \iff \langle s_0, \dots, s_n \rangle \in \mathcal{E} \wedge \langle s_n, \dots, s_m \rangle \in \mathcal{E}$. This assumption is required for the derivation of computable approximations of dependences. It is satisfied for all semantic slices proposed in [23] (a little complication arises in the case of the "pattern"-based semantic slicing: in this case, the dependence analysis should use the same partitioning criteria as the static analysis that computes $\mathcal{E}^\sharp$; this case is evoked in Sect. 4).

Example 3 (Semantic slicing). Fig. 2 presents some cases of semantic slices defined by a set of initial and final states. In the case of Fig. 2(a) (similar to Ex. 2), the condition on the input b entails that only the true branch may be executed; moreover, the value of b may not change (it is equal to **true**), so we expect all dependences on (l_0, b) be removed.

Similarly, in the case of Fig. 2(b), the output condition on y may only be achieved by executions flowing through the true branch of the conditional; therefore, we expect the dependences of (l_5, y) on $(l_0, z), (l_0, y)$ not to be considered.

In the same way as in Sect. 2.2, we propose a definition for dependences induced by functions (the definition for dependences induced by a program follows straightforwardly). More precisely, we consider in the following definition the case of a function ϕ constrained by input and output conditions; the case of functions abstracting a program semantic slice is more technical but similar.

Definition 2 (Observable dependences). *Let* $\phi \in \mathfrak{Den}$, $\mathcal{M}_i, \mathcal{M}_o \subseteq \mathbb{M}$, $x_0, x_1 \in \mathbb{X}$. *We say that* ϕ *induces an* observable dependence *of* x_1 *on* x_0 *in the semantic slice* $(\mathcal{M}_i, \mathcal{M}_o)$ *if and only if* $\exists \rho \in \mathcal{M}_i,\ v_a, v_b \in \mathcal{M}_i(x_0)$ *and such that* $\phi(\rho[x_0 \leftarrow v_a])(x_1) \cap \mathcal{M}_o(x_1) \neq \phi(\rho[x_0 \leftarrow v_b])(x_1) \cap \mathcal{M}_o(x_1)$. *We write* $x_1 \overset{\phi}{\rightsquigarrow}_{\mathcal{M}_i \Rightarrow \mathcal{M}_o} x_0$ *for such a dependence.*

Intuitively, an observable dependence is a dependence of *the original function*, which can be observed even when considering executions and values in the slice only. This notion generalizes the classical dependences presented in Def. 1: indeed, if we let $\mathcal{M}_i = \mathcal{M}_o = \mathbb{M}$, we find the same notion as in Def. 1.

Other possible definitions for observable dependences could have been chosen; however, most of them are flawed. For instance, considering the dependences of the restriction $\widetilde{\phi} : \rho \mapsto \phi(\{\rho\} \cap \mathcal{M}_i) \cap \mathcal{M}_o$ would have caused many additional dependences, with no intuitive interpretation: slicing $\phi = \lambda\rho.\{\rho\}$ with the input condition $\rho(x) = 0$ would include dependences of the form $y \overset{\tilde{\phi}}{\rightsquigarrow} x$ for *any* variable y, which would not be meaningful. Therefore, we consider dependences of ϕ, observed on the restriction.

Example 4 (Ex. 3 continued). Let us consider the program in Fig. 2(a). There is only one possible value for b at l_0, so Def. 2 defines no dependence on (l_0, b), as expected in Ex. 3.

In the program of Fig. 2(b), the condition on the output rules out all traces going through the false branch. As a consequence, the set of dependences in the semantic slice between l_0 and l_2 is $\{(y, x)\} \cup \{(z, z) \mid z \in \mathbb{X},\ z \neq y\}$.

2.4 Abstract Dependences

A second restriction consists in defining dependences among *abstractions* of the values the variables may take in the semantic slice. We consider simple abstractions only. For instance, we may wish to find out what may cause some variable to take large values (e.g., to investigate an overflow alarm), or very small values (e.g., to investigate a division by 0), or out-of-spec values (in case a user-provided specification maps variables to ranges they are supposed to live in). We let such a property be represented by an abstraction of sets of values, defined by a Galois-connection $\mathcal{P}(\mathbb{V}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} D$. The formalism of Galois-connections is powerful enough for our needs here, since we express dependences among simple abstractions only (i.e. α is always defined). We let $\mathbb{A}$ denote the set of such abstractions. An abstraction will be identified to its abstraction function, since there is no ambiguity. For instance, if k is a large scalar value, we may define $\gamma^{[k]}(\mathscr{P}_{\forall}^{[k]}) = \{v \mid |v| < k\}$, $\gamma^{[k]}(\mathscr{P}_{\exists}^{[k]}) = \{v \mid |v| \geq k\}$ and $\mathbb{D}^{[k]} = \{\bot, \mathscr{P}_{\forall}^{[k]}, \mathscr{P}_{\exists}^{[k]}, \top\}$; this abstraction allows to select which variable may take a large value. If the analyzer reports a possible overflow of x, then, we may wish to check what x depends on, and more precisely what variables may take abnormal or special (e.g., large) values, causing x to overflow; indeed, most arithmetic operators (like $+$, $-$, $\star$) tend to propagate large values in concrete executions and abstract analyses. For instance, we may want to learn what may cause y to grow above 1 000 in the program in Fig. 2(b) (this was the purpose of the output condition on y when defining the semantic slice) and more precisely to track other abnormal values in the computation of y. This is the goal of the following definition:

Definition 3 (Abstract dependences). *Let $\phi \in \mathfrak{Den}$, $\mathcal{M}_i, \mathcal{M}_o \subseteq \mathbb{M}$, $x_0, x_1 \in \mathbb{X}$, $\alpha_0, \alpha_1 \in \mathbb{A}$ (we write D_0, D_1 for the abstract domains corresponding to α_0, α_1). We say that ϕ induces an* abstract dependence *of (x_1, α_1) on (x_0, α_0) in the semantic slice $(\mathcal{M}_i, \mathcal{M}_o)$ if and only if $\exists \rho \in \mathcal{M}_i$, $d_a, d_b \in D_0$ and such that:*

- *$\forall j \in \{a, b\}, \gamma_0(d_j) \cap \mathcal{M}_i(x_0) \neq \emptyset$;*
- *$\alpha_1(\phi(\rho[x_0 \leftarrow \gamma_0(d_a)])(x_1) \cap \mathcal{M}_o(x_1)) \neq \alpha_1(\phi(\rho[x_0 \leftarrow \gamma_0(d_b)])(x_1) \cap \mathcal{M}_o(x_1))$.*

We write $(x_1, \alpha_1) \rightsquigarrow_{\mathcal{M}_i \mapsto \mathcal{M}_o} (x_0, \alpha_0)$ for such a dependence.

Intuitively, an abstract dependence is a dependence that can be observed by looking at abstractions of the values of the variables only. In particular, we can remark that the notion of abstract dependences generalizes the notion of observable dependences: if we let $\alpha_0 = \alpha_1 = \mathbf{id}$ where $\forall P \subseteq \mathbb{M}$, $\mathbf{id}(P) = P$, we define the same notion as in Def. 2. As usual, this definition is implicitly extended to dependences in programs.

Example 5 (Ex. 3 continued). We consider the dependences of (l_5, y) on the example of Fig. 2(b) again, but we wish to consider dependences involving "large" values only, i.e. we consider abstractions of the form $\alpha^{[k]}$ where $k > 1\,000$, with the above notations. Since x does not take any large value, the dependence of $(l_5, y, \alpha^{[1\,000]})$ is restricted to $(l_2, y, \alpha^{[1\,000]})$. Furthermore, in this case, the first occurrence of a large value in the program coincides with the assignment right before l_2; in this sense, following the abstract dependence allows to get an insight about where the abnormal value for y comes from.

Clearly, the approach proposed here may not lead to the actual error behind an alarm (e.g., an overflow). First, some large values may be caused by a division by small values (this case requires considering abstract dependences involving various kind of abstractions). Second an overflow may be due to a slow divergence; in this case, only the "cycle" of dependences corresponding to the diverging values will be discovered. Ideally, we would look for dependences of the form $(x_1, \alpha_1) \rightsquigarrow (x_0, \mathbf{id})$ in order to collect all dependences of a variable x_1 causing an overflow; yet this would tend to yield too many dependences. Our approach mainly aims at characterizing which variables are more likely to cause an error, by looking at the dependences that may carry abnormal (e.g. large) values first.

3 Comparing Dependences

In this section, we show in what extent abstract and observable dependences are stronger forms of dependences than the standard notion presented in Def. 1.

Theorem 1 (Hierarchy of dependences). *We let $\phi \in \mathfrak{Den}$, $\mathcal{M}_i, \mathcal{M}_o \subseteq \mathbb{M}$, $x_0, x_1 \in \mathbb{X}$, $\alpha_0, \alpha_1 \in \mathbb{A}$. Then:*

- *if ϕ induces a dependence $x_1 \rightsquigarrow_{\mathcal{M}_i \Rightarrow \mathcal{M}_o} x_0$, and $\mathcal{M}_i', \mathcal{M}_o'$ are such that $\mathcal{M}_i \subseteq \mathcal{M}_i'$ and $\mathcal{M}_o \subseteq \mathcal{M}_o'$, then ϕ induces a dependence $x_1 \rightsquigarrow_{\mathcal{M}_i' \Rightarrow \mathcal{M}_o'} x_0$;*
- *if ϕ induces a dependence $(x_1, \alpha_1) \rightsquigarrow_{\mathcal{M}_i \Rightarrow \mathcal{M}_o} (x_0, \alpha_0)$, and α_0' is less abstract than α_0 (i.e., there exists an abstraction α_0'' such that $\alpha_0 = \alpha_0'' \circ \alpha_0'$) and α_1' is less abstract than α_1, then ϕ induces a dependence $(x_1, \alpha_1') \rightsquigarrow_{\mathcal{M}_i \Rightarrow \mathcal{M}_o} (x_0, \alpha_0')$ (in particular, if we let $\alpha_0 = \alpha_1 = \mathbf{id}$, then we conclude that there exists a dependence $x_1 \rightsquigarrow_{\mathcal{M}_i \Rightarrow \mathcal{M}_o} x_0$);*
- *in particular, if ϕ induces a dependence $(x_1, \alpha_1) \rightsquigarrow_{\mathcal{M}_i \Rightarrow \mathcal{M}_o} (x_0, \alpha_0)$, and if we let $\mathcal{M}_i' = \mathcal{M}_o' = \mathbb{M}$ and $\alpha_0 = \alpha_1 = \mathbf{id}$, then we conclude that there exists a dependence $x_1 \rightsquigarrow x_0$ in the sense of Def. 1.*

These properties are very intuitive: the smaller a semantic slice, the less dependences one can observe on it; similarly, the more "abstract" the abstractions we

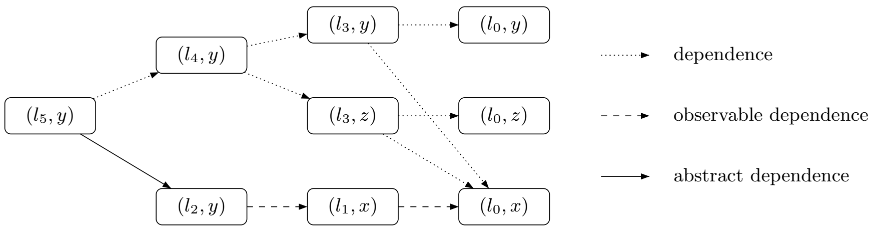


Fig. 3. Dependences from (l_5, y) in the program of Fig. 2(b)

consider, the less dependences they let observe (the abstractions may hide dependences). In particular, for any semantic slice and any abstraction, observable and abstract dependences are a subset of the "usual" dependences introduced in Def. 1. As a consequence, the abstract dependences allow to select *some* dependences, that are more likely to be useful when trying to understand the origin of (true or false) alarms; in this sense, they provide more precise information than mere dependences. We now apply these principles to the program of Fig. 2(b):

Example 6 (Ex. 5 continued). We present in Fig. 3 all possible kinds of *local* dependences (i.e.dependences on one-step transitions) collected recursively from (l_5, y). Next section discusses how to approximate dependences with such a graph. As shown in the figure, the restriction to observable dependences in a semantic slice defined by the conditions in Fig 2(a) allows to throw away the dependences induced by the **false** branch; the abstract dependences are even more restrictive, with only one abstract dependence. This dependence points to the assignment in the **true** branch where a large value is assigned to y.

4 Fixpoint-Based Approximation for Observable Dependences

At this point, we have introduced some relevant notions of dependences; yet, we need algorithms to compute them (exactly or with some approximation); the goal of this section is to provide a computable approximation for observable dependences, to compare it with existing methods and propose refinements. We generalize this techinique to the approximation of abstract dependences in the next section.

We start with a semantic-based fixpoint algorithm for approximating dependences and benefit from the semantic foundation to implement various refinements. The principle of this algorithm is comparable to existing dependence analyses [19]; yet, the advantage of our presentation is to allow for a wide variety of refinements inherited from static analysis to be formulated and proved; these refinements are described in the end of the section.

The approximation of composition: First, we propose to define an approximation for $\circ$ in $\mathfrak{Dep}_{\mathfrak{f}}$. We let $\phi_0, \phi_1 \in \mathfrak{Den}$ and consider the function

$\phi = \phi_1 \circ \phi_0$. We let $\mathcal{D}_0, \mathcal{D}_1$ be over-approximations of the dependences induced by ϕ_0, ϕ_1; we try to approximate the set $\mathcal{D}$ of dependences of ϕ. Let $(x_0, x_2) \in \mathbb{X}^2$, such that $\forall x_1 \in \mathbb{X},\ (x_0, x_1) \notin \mathfrak{D}_0 \vee (x_1, x_2) \notin \mathfrak{D}_1$. We let $\rho \in \mathbb{M}$, $v_a, v_b \in \mathbb{V}$, and $W = \{x_1 \in \mathbb{X} \mid \phi_0(\rho_a)(x_1) \neq \phi_0(\rho_b)(x_1)\}$ where $\forall i,\ \rho_i = \rho[x_0 \leftarrow v_i]$. We can prove by induction on the number of elements of W that $\phi_1 \circ \phi_0(\rho_a)(x_2) = \phi_1 \circ \phi_0(\rho_b)(x_2)$ (W is finite since $\mathbb{X}$ is finite). As a consequence:

Lemma 1 (Approximation of $\circ$). *With the above notations, $\mathcal{D} \subseteq \mathcal{D}_0 \boxdot \mathcal{D}_1$, where $\boxdot$ is the binary operator defined over $\mathfrak{Dep}_\mathrm{f}$ by $\mathcal{D}_0 \boxdot \mathcal{D}_1 = \{(x_0, x_2) \in \mathbb{X}^2 \mid \exists x_1 \in \mathbb{X},\ (x_0, x_1) \in \mathcal{D}_0 \wedge (x_1, x_2) \in \mathcal{D}_1\}$. As a consequence, if $\mathfrak{D}_\mathrm{f}[\phi_0] \subseteq \mathcal{D}_0$ and $\mathfrak{D}_\mathrm{f}[\phi_1] \subseteq \mathcal{D}_1$, then $\mathfrak{D}_\mathrm{f}[\phi_1 \circ \phi_0] \subseteq \mathcal{D}_0 \boxdot \mathcal{D}_1$.*

This approximation is clearly strict in general. Intuitively, the operator $\boxdot$ provides a sound approximation for $\circ$ in $\mathfrak{Dep}_\mathrm{f}$. An approximation for the dependences of semantic slices can be computed in a similar way. Let $\phi_0, \phi_1 \in \mathfrak{Den}$, and $\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2 \subseteq \mathbb{M}$, and $\phi = \phi_1 \circ \phi_0$. We consider the semantic slices of ϕ_0 and ϕ_1 defined respectively by $(\mathcal{M}_0, \mathcal{M}_1)$ and $(\mathcal{M}_1, \mathcal{M}_2)$; the semantic slice of the composition is $\widetilde{\phi} : \rho \mapsto \phi_1(\phi_0(\{\rho\} \cap \mathcal{M}_0) \cap \mathcal{M}_1) \cap \mathcal{M}_2$. If $\mathcal{D}_0, \mathcal{D}_1$ over-approximate the dependences of the semantic slices of ϕ_0 and of ϕ_1 respectively, then we can prove that $\mathcal{D}_0 \boxdot \mathcal{D}_1$ over-approximates the dependences of the slice $\widetilde{\phi}$.

Fixpoint-based over-approximation of dependences: The restriction $[\![s]\!]_{[p]}$ of $[\![s]\!]$ to a path $p = l_0 \cdot l_1 \cdot \ldots \cdot l_n$ is the set of traces that follow that path (i.e. of the form $\langle (l_0, \rho_0), (l_1, \rho_1), \ldots, (l_n, \rho_n) \rangle$). We can abstract $[\![s]\!]_{[p]}$ into a function $\mathfrak{f}_{[p]} \in \mathfrak{Den}$ defined by $\mathfrak{f}_{[p]} : \rho_0 \mapsto \{\rho_n \in \mathbb{M} \mid \langle (l_0, \rho_0), (l_1, \rho_1), \ldots, (l_n, \rho_n) \rangle \in [\![s]\!]\}$; furthermore, $\mathfrak{f}_{[p]} = \delta_{l_n}^{l_{n+1}} \circ \ldots \circ \delta_{l_0}^{l_1}$ where $\forall l, l' \in \mathbb{L},\ \delta_l^{l'}(\rho) = \{\rho' \in \mathbb{M} \mid (l, \rho) \rightarrow (l', \rho')\}$ is a local semantic transformer. At this point, we can make two remarks:

- Lemma 1 provides an approximation for the dependences induced by $\mathfrak{f}_{[p]}$: $\mathfrak{D}_\mathrm{f}[\mathfrak{f}_{[p]}] \subseteq \mathfrak{D}_\mathrm{f}[\delta_{l_0}^{l_1}] \boxdot \ldots \boxdot \mathfrak{D}_\mathrm{f}[\delta_{l_n}^{l_{n+1}}]$;
- the abstraction $\mathfrak{f}_{l_0}^{l_n}$ of the traces from l_0 to l_n can be decomposed along all paths from l_0 to l_n: $\forall \rho \in \mathbb{M},\ \mathfrak{f}_{l_0}^{l_n}(\rho) = \cup\{\mathfrak{f}_{[p]}(\rho) \mid p \text{ path from } l_0 \text{ to } l_n\}$; this allows to prove that a dependence between l_0 and l_n should be observable on at least one path from l_0 to l_n.

We let $\mathcal{D}_\mathrm{loc} \in \mathfrak{Dep}_\mathrm{t}$ be an approximation of all local dependences in s: $(x, x') \in \mathfrak{D}_\mathrm{f}[\delta_l^{l'}] \Rightarrow ((l, x), (l', x')) \in \mathcal{D}_\mathrm{loc}$. An example of a rough definition for $\mathcal{D}_\mathrm{loc}$ is shown on Fig. 4. We deduce from the two points above the following theorem (if F is montone, we write $\mathbf{lfp}_{x_0} F$ for the least fixpoint of F, greater than x_0):

Theorem 2 (Dependences approximation). *Let $\circledast$ be the operator defined on $\mathfrak{Dep}_\mathrm{t}$ by $\mathcal{D}_0 \circledast \mathcal{D}_1 = \{(\nu_0, \nu_2) \mid \exists \nu_1 \in (\mathbb{L} \times \mathbb{X}),\ (\nu_0, \nu_1) \in \mathcal{D}_0 \wedge (\nu_1, \nu_2) \in \mathcal{D}_1\}$, and $\Delta = \{(\nu, \nu) \mid \nu \in (\mathbb{L} \times \mathbb{X})\}$. Then, the dependences of s are approximated by:*

$$\mathfrak{D}_\mathrm{t}[s] \subseteq \mathbf{lfp}_\Delta \mathfrak{F}_\mathrm{dep} \qquad \textit{where } \mathfrak{F}_\mathrm{dep} : \mathfrak{Dep}_\mathrm{t} \rightarrow \mathfrak{Dep}_\mathrm{t};\ \mathcal{D} \mapsto \mathcal{D} \cup \mathcal{D}_\mathrm{loc} \circledast \mathcal{D}$$

This theorem provides a fixpoint-based algorithm for the over-approximation of dependences. Comparable algorithms can be obtained via typing approaches

$$\begin{array}{l} \mathit{use} : \mathbb{e} \to \mathcal{P}(\mathbb{X}) \\ \qquad \mathit{use}(c) = \emptyset \\ \qquad \mathit{use}(v) = \{v\} \\ \mathit{use}(e_0 \oplus e_1) = \mathit{use}(e_0) \cup \mathit{use}(e_1) \\ \forall e \in \mathbb{e},\ \rho \in \mathbb{M},\ x \in \mathbb{X},\ v_a, v_b \in \mathbb{V}, \\ \llbracket e \rrbracket(\rho_a) \neq \llbracket e \rrbracket(\rho_b) \Rightarrow x \in \mathit{use}(e) \\ (\text{where } \rho_i = \rho[x \leftarrow v_i]) \end{array}$$

(a) Deps. in expressions

$$\begin{array}{l} \text{assignment } l_0 : x = e; l_1 : \\ \mathfrak{D}_\mathfrak{f}[\delta_{l_0}^{l_1}] \subseteq \mathit{use}(e) \times \{x\} \cup \{(y, y) \mid y \in \mathbb{X} \setminus \{x\}\} \\ \text{loop } l_0 : \mathbf{while}(e)\{l_1 : \ldots\} \\ \mathfrak{D}_\mathfrak{f}[\delta_{l_0}^{l_1}] \subseteq \mathit{use}(e) \times \mathbb{X} \cup \{(y, y) \mid y \in \mathbb{X}\} \\ \text{conditional } l_0 : \mathbf{if}(e)\{l_1 : \ldots; l_2\}\mathbf{else}\{\ldots\}l_3 \\ \mathfrak{D}_\mathfrak{f}[\delta_{l_0}^{l_1}] \subseteq \{(x, y) \in \mathbb{X}^2 \mid x = y \vee x \in \mathit{use}(e)\} \\ \mathfrak{D}_\mathfrak{f}[\delta_{l_2}^{l_3}] \subseteq \{(x, x) \mid x \in \mathbb{X}\} \end{array}$$

(b) Approximation for local dependences

Fig. 4. Local dependences for a simple language

[1,2]; we prefer providing a fixpoint based definition in order to design various kinds of refinements (see the end of this section). Again this theorem also holds true in the case of observable dependences; however, the proof relies on the closeness assumption mentioned in Sect. 2.3. This hypothesis is necessary in order to prove the first point (decomposition of the dependences along a path).

Remark 1 (Control dependences). To simplify the presentation, the fixpoint algorithm of Theorem 2 does not distinguish control and data dependences like most dependence analyses do [19]. This would result in a loss of precision: for instance, in the case of a conditional $l_0 : \mathbf{if}(b)\{l_1\}l_2$, a dependence $(l_2, x) \rightsquigarrow (l_0, b)$ would be inferred for any variable x, since there is a dependence $(l_1, x) \rightsquigarrow (l_0, b)$. Yet, we can prove that, if $l, l' \in \mathbb{L}$ are such that $\forall \rho \in \mathbb{M},\ \mathfrak{f}_l^{l'}(\rho) \neq \emptyset$, if there exists a dependence $(l', x') \rightsquigarrow (l, x)$, then there exists a path from l to l' where the value of x is modified. In the above example, x is not modified between l_0 and l_1. Hence, our algorithm does not suffer the loss of precision mentioned above (our implementation does not include the fictitious dependence $(l_2, x) \rightsquigarrow (l_0, b)$).

Example 7 (Ex. 3 continued). We consider the program in Fig. 2(a) (the input condition is ignored here). Then, $\mathcal{D}_{\mathrm{loc}}$ contains the local dependences $(l_5, y) \rightsquigarrow (l_2, y)$, $(l_2, y) \rightsquigarrow (l_1, x), (l_1, x) \rightsquigarrow (l_0, b)$; the fixpoint algorithm of Theorem 2 composes these dependences together so, the dependence $(l_5, y) \rightsquigarrow (l_0, b)$ is discovered. The dependence $(l_5, y) \rightsquigarrow (l_0, x)$ is inferred in the same way. Obviously, the dependence on (l_0, b) does not hold in the semantic slice; so we show in the following how to get rid of it, by taking the properties of the semantic slice into account. Similarly, in the case of the program in Fig. 2(b), dependences through the false branch yield the dependences $(l_5, y) \rightsquigarrow (l_0, y), (l_0, z)$, which are not observable in the semantic slice.

Remark 2. Note that a sound dependence analysis for a real language (like C) requires sound aliasing information to be known: indeed, if x and y are aliased, an assignment to x creates an implicit dependence on y. Many alias analyses exist in the literature, e.g. [8,14], so we do not develop this issue here.

Dependence graphs: In general, we are not interested in *all* the dependences of s; we only wish to track the dependences of a *criterion* c, i.e. a set of pairs (control state,variable) of interest: the set of dependences of interest is $\mathit{dep}[c] =$

$\mathfrak{D}_t[s] \cap ((\mathbb{L} \times \mathbb{X}) \times c)$. For instance, in the case of Fig. 2(b), we considered the dependence of $\{(l_5, y)\}$. We can approximate $\mathit{dep}[c]$ by a least-fixpoint form:

Theorem 3 (Dependences of a criterion). $\mathit{dep}[c] \subseteq \mathbf{lfp}_{(\mathbb{L}\times\mathbb{X})\times c}\mathfrak{F}_{\mathrm{dep}}$

In practice, a superset of the dependences (i.e. of $\mathcal{D}_{\mathrm{loc}}$) is collected during a linear pass; then the computation of an over-approximation of the dependence of a criterion $c \subseteq \mathbb{L} \times \mathbb{X}$ follows from Theorem 3.

Refinements: We propose now a series of refinements, in order to restrict the local dependences and their global composition so as to carry out more precise fixpoint computations. These refinements can be expressed and proved formally on the basis of Theorem 2. We consider a semantic slice $\mathcal{E}$, approximated by $\mathcal{E}^\sharp : \mathbb{L} \to \mathcal{P}(\mathbb{M})$. Among these refinements, we can cite:

- **Removal of unreachable control states:** some control state $l \in \mathbb{L}$ may be unreachable in the semantic slice. In this case, it is obvious there can be no observable dependence from or to that point. In practice, the invariant $\mathcal{E}^\sharp$ computed in the semantic slicing phase [23] provides an over-approximation of the reachable control states in the semantic slice (if l reachable, then $\mathcal{E}^\sharp(l) \neq \emptyset$); any other control state should be removed from the dependences at this point.
- **Removal of constant variables:** similarly, a variable x may be proved constant at point l in the semantic slice by the analyzer (this amounts to proving $\exists v \in \mathbb{V},\ \mathcal{E}^\sharp(l)(x) \subseteq \{v\}$); in this case, there can be no dependence to (l, x): indeed, we cannot pick up two distinct values for x at l; as a consequence any two stores ρ_a, ρ_b differing at most in the value for x generate the same transitions from this point. For instance, in case the semantic slice specifies a constant value for some input variable, any variable computed from this input only is constant, hence should be removed from the dependences.
  Note that the same simplification on the other side of the dependence does not hold: indeed, proving $\rho(x) \subseteq \{v\}$ does not rule out that $\rho(x)$ may be $\emptyset$.
- **Simplification of constant expressions:** The above principle also applies to sub-expressions, which may help reducing the local dependences induced by assignments or conditions. For instance, if we consider the assignment $x = x_0 \star x_1 + x_1 \star x_2$, where $x, x_0, x_1, x_2 \in \mathbb{X}$ and x_0, x_1 are proved constant in the semantic slice, then only the dependence on x_2 should be considered.
- **Control partitioning:** the analysis carried out in the semantic slicing may resort to some kind of trace partitioning (either control-based [22] or to distinguish execution patterns [23]); then, the same principle could be applied to the dependence analysis. In particular, this approach allows to benefit from precise abstract invariants, so it may increase the number of contexts the above refinements can be applied to (for instance, some statements may be unreachable in *some* partitions, as shown in Ex. 9).

Example 8 (Ex. 7 continued). In Fig. 2(a), the value of b at l_0 is **true** (constant value) in the semantic slice; as a result, any dependence $(l_1, v) \rightsquigarrow (l_0, b)$ is removed, so that the dependence $(l_5, y) \rightsquigarrow (l_0, b)$ does not appear in the fixpoint computation anymore.

Similarly, in the semantic slice of the program in Fig. 2(b), the false branch of the conditional is unreachable; as a result any local dependence involving l_3 or l_4 is removed from $\mathcal{D}_{\mathrm{loc}}$; as a result, the dependences $(l_5, y) \rightsquigarrow (l_0, y), (l_0, z)$ are no longer computed.

Example 9 (Partitioning analysis). Let us consider the program $l_0 : \mathbf{if}(b)\{x_0 = y\}\mathbf{else}\{x_1 = y\}; \mathbf{if}(b')\{z = x_0\}\mathbf{else}\{z = x_1\}; l_1$ and the semantic slice collecting all executions going through the *same* branch in both **if** statements. Then, the partitioning dependence analysis infers only one dependence from (l_1, z), namely (l_0, y). The non-partitioning analysis would also include dependences on $(l_0, b), (l_0, b'), (l_0, x_1), (l_0, x_0)$. We can see that this refinement allows for global precision improvements.

5 Approximating Abstract Dependences

Chains of abstract dependences: All results of Sect. 4 can be generalized straightforwardly to the case of abstract dependences. In particular, Lemma 1 and Theorem 2 can be generalized, by taking the abstractions into account in the definition of $\boxdot$ and $\boxast$. Indeed, we could prove as for Lemma 1 that $((x_0, \alpha_0), (x_2, \alpha_2)) \in \mathfrak{D}_{\mathrm{f}}[\phi_1 \circ \phi_0]$ entails that there exists $(x_1, \alpha_1) \in \mathbb{X} \times \mathbb{A}$ such that $((x_0, \alpha_0), (x_1, \alpha_1)) \in \mathfrak{D}_{\mathrm{f}}[\phi_0]$ and $((x_1, \alpha_1), (x_2, \alpha_2)) \in \mathfrak{D}_{\mathrm{f}}[\phi_1]$.

However, this solution is not completely satisfactory for several reasons:

- the lattice of all abstractions of $\mathcal{P}(\mathbb{V})$ is not representable.
- the fixpoint-based expressions would lead to a rough approximation. In particular if $(l_2, x_2, \alpha_2) \overset{\phi_1}{\rightsquigarrow} (l_1, x_1, \alpha_1)$ and $(l_1, x_1, \alpha_1) \overset{\phi_0}{\rightsquigarrow} (l_0, x_0, \alpha_0)$, then a dependence $(l_2, x_2, \alpha_2) \overset{\phi_1 \circ \phi_0}{\rightsquigarrow} (l_0, x_0, \alpha_0)$ will always be added, which is overly conservative in the case of abstract dependences.
- we wish to compute sets of abstract dependences that are *immediately* relevant to the criterion; indeed, given a criterion $c = (l, x, \alpha)$, we would like to track the observable abstract dependences following immediately from c first; more complex dependences should be considered only after the simpler ones did not reveal relevant causes for the alarm under investigation. In this sense, an *under*-approximation of the abstract dependences from the criterion makes sense.

As a consequence, we introduce a notion of abstract dependence chain, which collects local abstract dependences, involving "interesting" abstractions only:

Definition 4 ($\mathbb{a}$-abstract dependence chain). *We let $\mathbb{a} \subseteq \mathbb{A}$ be a set of abstractions of interest and c be the criterion (l, x, α). An* $\mathbb{a}$-abstract dependence chain *from c is a finite sequence $(l_0, x_0, \alpha_0), \ldots, (l_n, x_n, \alpha_n)$, such that:*

1. *$\forall i,\ \alpha_i \in \mathbb{a}$,*
2. *$\forall i,\ ((l_i, x_i, \alpha_i), (l_{i+1}, x_{i+1}, \alpha_{i+1})) \in \mathcal{D}_{\mathrm{loc}}$.*

For instance, we may choose a family of abstractions composed of the abstractions mentioned in Sect. 2.4; e.g., we may let $\mathbb{a} = \{\alpha^{[10^n]} \mid n \in \mathbb{N},\ n \geq 3\}$, so as to track large values.

Computation of abstract dependence chains: We need an *abstract dependence graph*, that is, an over-approximation for all abstract dependences involving abstractions in $\mathbb{a}$ only, that occur on one-step transitions (that is, on edges of the control flow graph). The rules defined in Sect. 2.4 apply for the over-approximation of such dependences; refinements of these local dependences are considered below. In practice, the representation of the abstract dependence graph consists in a dependence graph, with labels on the edges, that approximate the abstractions the dependences they correspond to are valid for.

Once the abstract dependence graph is computed, an over-approximation of the $\mathbb{a}$-abstract dependence chains from any criterion $c \in \mathbb{L} \times \mathbb{X} \times \mathbb{A}$ can be computed as suggested by Theorem 3, by a fixpoint-based algorithm.

Refinements: All refinements introduced in the case of (concrete) observable dependences, in Sect. 2.4 are also sound in the case of abstract dependences.

We propose a refinement that generalizes the "removal of constant variables" (Sect. 4) to abstract dependences. Let us consider $(l_0, x_0, \alpha_0), (l_1, x_1, \alpha_1) \in \mathbb{L} \times \mathbb{X} \times \mathbb{A}$. If there exists a minimal element d_0 of $D_0 \setminus \{\perp\}$ (where $\perp$ is the least element of D_0) such that $\mathcal{E}^\sharp(l_0)(x_0) \subseteq \gamma_0(d_0)$, then the abstract domain D_0 is not able to distinguish the values observed for x_0 at l_0 in the semantic slice. An obvious application of Def. 3 shows that there is no dependence $(l_1, x_1, \alpha_1) \overset{\mathcal{E}}{\rightsquigarrow} (l_0, x_0, \alpha_0)$. For instance, this refinement applies if α_0 abstracts together all "normal" (i.e., not too large) values and if all values for x_0 at point l_0 are "normal".

Example 10 (Ex. 5 continued). For instance, in the case of the program in Fig. 2(b), $x \in [0, 10]$ at point l_0; hence, if we consider $\mathbb{a}$ as defined above, there exist no abstractions $\alpha_x, \alpha_y \in \mathbb{a}$ such that $(l_2, y, \alpha_y) \overset{\mathcal{E}}{\rightsquigarrow} (l_1, x, \alpha_x)$. As a consequence, the only remaining abstract dependence from (l_5, y) in the semantic slice and involving abstractions in $\mathbb{a}$ is a dependence of (l_5, y) on (l_2, y); this $\mathbb{a}$-abstract dependence chain leads to the point where an "abnormal" value appears for the first time in the sequence of computations leading to y (see Fig. 3).

6 Slicing and Case Study

Slicing: Slicing [26] aims at selecting a subset of the statements of a program that may play a role in the computation of some variable x at some point l. The principle is to include in the slice any statement at point l' that may modify a variable x' such that (l, x) depends on (l', x').

The semantics of program slicing is rather subtle for several reasons:

- The notion of dependence involved in slicing is quite different to the one we considered in Sect. 2. For instance the slice of $l_0 : x = 3; l_1 : y = x; l_2$ for the criterion (l_2, y) should include the statement $l_0 : x = 3; l_1$ as well, even though (l_2, y) does not depend on (l_1, x) according to Def. 2, since x is constant at l_1.

– The usual expression of slicing correctness resorts to some kind of projection of the program semantics, which is preserved by slicing. However, the removal of non-terminating loops (or of possible sources for errors) may cause the slice to present *more* behaviors than the projection of the semantics of the source program. This issue can be solved by considering a non-standard, more concrete semantics [7], which is preserved by the transformation, yet this approach is not natural for static analysis.

As a consequence, we propose a transformation that should be more adapted to static analysis.

Smaller, non-executable slices: In [23], semantic slices approximate program executions with abstract invariants. Such an invariant together with a (subset of a) syntactic slice allow to describe even more precisely a set of program executions:

Definition 5 (Abstract slice). *An abstract slice $\mathcal{E}$ of a program s is defined by a sound invariant $\mathcal{E}^\sharp : \mathbb{L} \to \mathcal{P}(\mathbb{M})$ for $\mathcal{E}$ and a subset s' of the program statements, which is defined by the set of corresponding control states $\mathcal{L}'$.*

The semantics of a semantic slice is defined both by the program transitions (for the statements which are included in the slice) and by the abstract invariants:

Definition 6 (Abstract slice semantics). *The semantics $(\!|s'|\!)$ of the abstract slice collects all the traces $\langle (l_0, \rho_0), \ldots, (l_n, \rho_n) \rangle$ such that:*
– $\forall i,\ \rho_i \in \mathcal{E}^\sharp(l_i)$;
– $\forall i,\ (l_i \in \mathcal{L}' \wedge l_{i+1} \in \mathcal{L}') \Longrightarrow (l_i, \rho_i) \to (l_{i+1}, \rho_{i+1})$.

Obviously, the definition of abstract slices leaves the choice of the syntactic slice undetermined. However, the purpose of the abstract slices is to restrict to the most interesting parts the program; hence, we propose to compute abstract dependence chains and include any assignment which affect a variable in a dependence chain: this way, the slice preserves only the α-abstract dependence chains and abstract any other statement of the program into the invariants in $\mathcal{E}^\sharp$. Let us note that this notion allows to solve the two points mentioned above:

– parts of the program that are not immediately relevant to the criterion under investigation (in the sense that they do not appear in the dependences introduced in Def. 1, Def. 2 and Def. 3) do *not* need to be included into the slice anymore; instead, they can be replaced with program invariants (in the semantic slice). For instance, the assignment $l_0 : x = 3; l_1$ can be replaced with the invariant $x = 3$ at point l_1. Obviously, applying this principle to larger programs may result in huge gain in slice sizes.
– the intersection with program invariants limits the loss of precision induced by, e.g. the removal of a loop.

Example 11 (Abstract slice). Let us consider the program of Fig. 2(b), together with its input/output conditions. Fig. 3 displays the local, observable and abstract dependences that can be recursively composed when starting from (l_5, y).

In case we compute an abstract slice for this program, starting from (l_5, y), we find only one $\mathbb{a}$-abstract dependence chain (Ex. 10). As a consequence, we get the abstract slice defined by the set of control states $\mathcal{L}' = \{l_1, l_2, l_5\}$. In particular, the abstract slice contains the assignment $l_1 : y = 1000 \star x; l_2$, with the invariant $(x \in [5, 10])$, which gives a likely cause for the error.

Early implementation results and case studies: A simple abstract dependence analysis was implemented inside ASTRÉE (for tracking large values and overflows), together with an abstract slice extraction algorithm. We could run these algorithms on some 70 kLOC real world program, which we modified so as to make some computations unstable (ASTRÉE proves the absence of overflow in the original version). The static analysis by ASTRÉE takes roughly 20 minutes and uses 500 Mb on a Bi-opteron 2.2 Ghz with 8 Gb of RAM. The computation of the dependence graph (by collecting all local dependences and applying local refinements) takes 72 seconds and requires 300 Mb, on the same machine; this phase provides all data required to extract a slice from any criterion. The slice extraction computes a least fixpoint from the criterion (Theorem 3) and applies recursively local dependences; in the case of abstract dependences, this amounts to collecting $\mathbb{a}$-abstract dependence chains. The typical slice extraction time is about 5 seconds, with low memory requirements (around 110 Mb).

The table below displays the gain in size obtained by computing abstract slices for a series of alarms (size of slices are in LOCs):

Slicing point	a_1	a_2	a_3
Classical slice	543	368	1572
Abstract slice	39	160	96

The resulting slices proved helpful for finding the direct consequences of errors like overflows; moreover, it seemed promising for deriving automatically semantic slicing criteria, which was one of the motivations for our present work. We remarked that the refinements presented in Section 4 played a great role in keeping the size of dependences down. Cyclic abstract dependence chains suggest some kind of partitioning could be done in order to isolate certain execution patterns; they also allow to restrict the part of the program to look at in order to define an adequate input for defining an error scenario, so that we envisage synthesizing input constraints in the future. Another possible use for abstract slices is to cut down the size of programs to analyze during alarm inspection sessions, by abstracting into invariants parts of the code to analyze.

7 Conclusion and Related Work

We proposed a framework for defining and computing valuable dependence information, for the understanding and refinement of static analysis results. Early experiments back-up favorably the usefulness of this approach, especially for giving good hints for the choice of semantic slicing criteria [23].

Our definition for dependences are rather related to the definition of non-interference [18] commonly used in language-based security [24]. This approach is rather different to the more traditional ways of defining dependences in program slicing, which rely on program dependence graphs [19], yet these two problems are related [2,1]. We found that the main benefit of the "dependences as interference" definition is to allow for wide varieties of refinements for dependence analyses and extension for the definition of dependences to be stated.

In particular, our definition of abstract dependences is closely related to the notion of abstract non interference introduced in [17] in the security area, which aims at classifying program attackers as abstract-interpretations. The authors propose to compute the strongest safe attacker of a program by resolving an equation on domains by fixpoint. In our settings, the abstraction on the output is fixed by the kind of alarm being investigated; moreover, the dependence analysis should discover the variables the criterion depends on and not only for what observation. Therefore, the algorithms proposed in [17] do not apply to our goal.

Program slicing [26] is another area related to our work. Many alternative notions of slices have been proposed since the first, syntactic versions of slicing. In particular, conditioned slicing [6] (applied, e.g. in [12]) aim at extracting slices preserving *some* executions of programs, specified by, e.g. a relation on inputs. Our approach goes beyond these methods: indeed, a set of program executions defined by a semantic property (e.g. leading to an error) is characterized precisely by semantic slicing [23]; these invariants allow to refine precisely the dependences. Dynamic slicing [3,20,16] records states during *concrete* executions and inserts a dependence among the corresponding nodes according to a standard, rough dependence analysis, in order to produce "dynamic", non-executable slices. This approach is adapted to debugging; yet it does not allow to characterize precisely a set of executions defined by semantic constraints either.

There exist a wide variety of methods applied to error cause localization. For instance, [4] proposes to characterize transitions that *always* lead to an error in abstract models; however, this kind of approach requires enumerating the predicates and/or transitions; hence, it does not apply to ASTRÉE, due to the number of predicates in the abstract invariants (domains nearly infinite).

Debugging methods start with a *concrete* trace, which we precisely do not have, since alarms arise from abstract analyzes.

A first possible direction for future work would be to express abstract dependences involving more complicated, e.g. relational abstractions. Indeed, tracking the origin of an alarm raised in the analysis of $z = \sqrt{x+y}$ requires looking at dependences involving the property $x + y < 0$. A second challenge is to let the dependence analysis interact more closely with the forward-backward analyses carried out by the semantic slicer [23]; in particular the dependence information could give some hints about what part of the invariants to refine (after specializing the semantic slicing criteria).

Acknowledgments. We deeply thank B. Blanchet and J. Feret for their comments on a preliminary version of this article.

References

1. M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 1999.
2. M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL*, 1999.
3. H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI*, 1990.
4. T. Ball, M. Naik, and S. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL*, 2003.
5. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety Critical Software. In *PLDI*, 2003.
6. G. Canfora, A. Cimitille, and A. D. Lucia. Condition program slicing. *Information and Software Technology; Special issue on Program Slicing*, 1998.
7. R. Cartwright and M. Felleisen. The semantics of program dependence. In *PLDI*, 1989.
8. J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *PLDI*, 1993.
9. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *ENTCS*, 6, 1997.
10. P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
11. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP*, 2005.
12. S. Danicic, D. Daoudi, C. Fox, R. Hierons, M. Harman, J. Howroyd, L. Ouarbya, and M. Ward. ConSUS: A Light-Weight Program Conditioner. *Journal of Systems and Software*, 2004.
13. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 1976.
14. A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI*, 1994.
15. N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI*, 2003.
16. C. Fox, S. Danicic, M. Harman, and R. Hierons. ConSIT: A Conditioned Program Slicing System. *Software - Practice and Experience*, 2004.
17. R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *POPL*, 2004.
18. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, 1982.
19. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI*, 1988.
20. B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 1988.
21. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, 2000.
22. L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *ESOP*, 2005.
23. X. Rival. Understanding the origin of alarms in ASTRÉE. In *SAS*, 2005.
24. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.
25. A. Venet and G. Brat. Precise and efficient array bound checking for large embedded c programs. In *PLDI*, 2004.
26. M. Weiser. Program slicing. In *Proceeding of the Fifth International Conference on Software Engineering*, pages 439–449, 1981.

A Typed, Compositional Logic for a Stack-Based Abstract Machine

Nick Benton

Microsoft Research, Cambridge
nick@microsoft.com

Abstract. We define a compositional program logic in the style of Floyd and Hoare for a simple, typed, stack-based abstract machine with unstructured control flow, global variables and mutually recursive procedure calls. Notable features of the logic include a careful treatment of auxiliary variables and quantification and the use of substructural typing to permit local, modular reasoning about program fragments. Semantic soundness is established using an interpretation of types and assertions defined by orthogonality with respect to sets of contexts.

1 Introduction

Recent research on language-based techniques in security and certification has led to renewed interest in Floyd-Hoare and VDM-style programming logics, and to much work on type systems and logics for low-level code. Two industrially significant typed intermediate languges have received a great deal of attention: the bytecode of the JVM, used as a target for Java, and the Common Intermediate Language of the CLR, used as a target for languages including $C^\sharp$ and Visual Basic. Both of these intermediate languages are stack-based with control flow expressed using labelled jumps and method calls.

Most research on formalizing the type systems of these intermediate languages [33, 12] has treated the reality of stacks and jumps, though some authors have chosen to work with structured imperative control flow [13] or functional-style applicative expressions [36]. Work on more general specification logics [1, 28, 16] has, however, mostly been done in the context of high-level languages.

Here we present and prove the correctness of a simple logic for directly proving partial correctness assertions on a minimal stack-based machine with jumps and first-order procedure calls. This is rather more complex than traditional Hoare logic for while programs. As well as unstructured control flow, we have to deal with a stack that varies in size and locations that vary in type, which means some care has to be taken to ensure assertions are even well-formed. There are also various kinds of error that are, at least a priori, possible in the dynamic semantics: stack underflow, wild jumps and type errors. We deal with these issues by defining a fairly simple type system that rules out erroneous behaviour, and defining assertions relative to typed programs.

There are also complexities associated with (possibly mutually-recursive) procedure calls, which become especially acute if one wishes to be able to reason locally and modularly, rather than re-analysing bodies at every callsite. We solve these problems using three techniques: firstly, we are very explicit about types, contexts and quantifiers (in particular, we have universally quantified assertions on labels in the context, in the style of Reynolds's specification logic [31]); secondly, we adopt a 'tight' interpretation of store typings, which allows us to use substructural reasoning to adapt assumptions on procedures to their calling context; thirdly, we use a rather general rely/guarantee-style rule for linking arbitrary program fragments.

The other novelty is the semantics with respect to which we prove soundness. Assertions on a program p are interpreted extensionally, using a form of orthogonality (perping) with respect to contexts extending p. A further twist in the proofs is the use of step-indexed approximations to the semantics of assertions and their orthogonals. Fuller details, including proofs, may be found in the companion technical report [8].

2 The Machine

The metavariables n and b range over the integers, $\mathbb{Z}$, and booleans, $\mathbb{B}$, respectively. We assume a set $\mathbb{V}$ of names for global variables, ranged over by x. The metavariables bop and uop range over typed (binary and unary, respectively) arithmetic and logical operations such as addition and conjunction. Programs, p, are finite partial functions from labels, $l \in \mathbb{N}$, to instructions I:

$$\begin{aligned} I &:= \texttt{pushc}\ \underline{v} \mid \texttt{pushv}\ \texttt{x} \mid \texttt{pop}\ \texttt{x} \mid \texttt{dup} \mid \texttt{binop}_{bop} \mid \\ &\quad\ \texttt{unop}_{uop} \mid \texttt{brtrue}\ l \mid \texttt{call}\ l \mid \texttt{ret} \mid \texttt{halt} \\ \mathit{Programs} \ni p &:= [l_1 : I_1, \ldots, l_n : I_n] \end{aligned}$$

Stores are finite functions from $\mathbb{V}$ to values (i.e. to $\mathbb{B} \cup \mathbb{Z}$). Our machine has two stacks: the evaluation stack, σ, used for intermediate values, passing arguments and returning results, is a finite sequence of values, whilst the control stack, C, is a finite sequence of labels (return addresses):

$$\begin{aligned} \mathit{Stores} \ni G &:= x_1 = v_1, \ldots, x_n = v_n \\ \mathit{Stacks} \ni \sigma &:= v_1, \ldots, v_n \\ \mathit{Callstacks} \ni C &:= l_1, \ldots, l_n \end{aligned}$$

We use a comma ',' for both the noncommutative, total concatenation of sequences and for the commutative, partial union of finite maps with disjoint domains. We write a dash '$-$' for both the empty sequence and the empty finite map, and use $|\cdot|$ for the length operation on finite sequences. A configuration is quintuple of a program, a callstack, a global store, an evaluation stack and a label (the program counter):

$$\mathit{Configs} = \mathit{Programs} \times \mathit{Callstacks} \times \mathit{Stores} \times \mathit{Stacks} \times \mathbb{N}$$

The operational semantics is defined by the small-step transition relation $\rightarrow$ on configurations shown in Figure 1. The $\texttt{pushc}\ \underline{v}$ instruction pushes a constant boolean or integer value v onto the evaluation stack. The $\texttt{pushv}\ \texttt{x}$ instruction pushes the value of the variable x onto the stack. The $\texttt{pop}\ \texttt{x}$ instruction pops the top element off the stack

$$\begin{array}{rl}
\langle p, l : \texttt{pushc}\ v|C|G|\sigma|l\rangle & \rightarrow \langle p, l : \texttt{pushc}\ v|C|G|\sigma, v|l+1\rangle \\
\langle p, l : \texttt{pushv}\ \mathrm{x}|C|G, x = v|\sigma|l\rangle & \rightarrow \langle p, l : \texttt{pushv}\ \mathrm{x}|C|G, x = v|\sigma, v|l+1\rangle \\
\langle p, l : \texttt{dup}|C|G|\sigma, v|l\rangle & \rightarrow \langle p, l : \texttt{dup}|C|G|\sigma, v, v|l+1\rangle \\
\langle p, l : \texttt{pop}\ \mathrm{x}|C|G, x = v'|\sigma, v|l\rangle & \rightarrow \langle p, l : \texttt{pop}\ \mathrm{x}|C|G, x = v|\sigma|l+1\rangle \\
\langle p, l : \texttt{binop}_{bop}|C|G|\sigma, v_1, v_2|l\rangle & \rightarrow \langle p, l : \texttt{binop}_{bop}|C|G|\sigma, v_3|l+1\rangle \\
\text{if } v_3 = (v_1\ bop\ v_2). & \\
\langle p, l : \texttt{unop}_{uop}|C|G|\sigma, v|l\rangle & \rightarrow \langle p, l : \texttt{unop}_{uop}|C|G|\sigma, v'|l+1\rangle \\
\text{if } v' = uop\ v. & \\
\langle p, l : \texttt{brtrue}\ l'|C|G|\sigma, true|l\rangle & \rightarrow \langle p, l : \texttt{brtrue}\ l'|C|G|\sigma|l'\rangle \\
\langle p, l : \texttt{brtrue}\ l'|C|G|\sigma, false|l\rangle & \rightarrow \langle p, l : \texttt{brtrue}\ l'|C|G|\sigma|l+1\rangle \\
\langle p, l : \texttt{call}\ l'|C|G|\sigma|l\rangle & \rightarrow \langle p, l : \texttt{call}\ l'|C, l+1|G|\sigma|l'\rangle \\
\langle p, l : \texttt{ret}|C, l'|G|\sigma|l\rangle & \rightarrow \langle p, l : \texttt{ret}|C|G|\sigma|l'\rangle
\end{array}$$

Fig. 1. Operational Semantics of the Abstract Machine

and stores it in the variable x. The $\texttt{binop}_{op}$ instruction pops the top two elements off the stack and pushes the result of applying the binary operator *op* to them, provided their sorts match the signature of the operation. The `brtrue` l instruction pops the top element of the stack and transfers control either to label l if the value was $true$, or to the next instruction if it was $false$. The `halt` instruction halts the execution. The `call` l instruction pushes a return address onto the call stack before transferring control to label l. The `ret` instruction transfers control to a return address popped from the control stack.

For $k \in \mathbb{N}$, we define the k-step transition relation $\rightarrow^k$ and the infinite transition predicate $\rightarrow^\omega$ in the usual way. We say a configuration is 'safe for k steps' if it either halts within k steps or takes k transitions without error. Formally:

$$\mathit{Safe}_0\langle p|C|G|\sigma|l\rangle \qquad\qquad \mathit{Safe}_k\langle p, l : \texttt{halt}|C|G|\sigma|l\rangle$$

$$\frac{\langle p|C|G|\sigma|l\rangle \rightarrow \langle p|C'|G'|\sigma'|l'\rangle \quad \mathit{Safe}_k\langle p|C'|G'|\sigma'|l'\rangle}{\mathit{Safe}_{k+1}\langle p|C|G|\sigma|l\rangle}$$

and we write $\mathit{Safe}_\omega\langle p|C|G|\sigma|l\rangle$ to mean $\forall k \in \mathbb{N}. \mathit{Safe}_k\langle p|C|G|\sigma|l\rangle$.

Although this semantics is fairly standard, the choice to work with partial stores is significant: execution can get stuck accessing an undefined variable, so, for example, there are contexts which distinguish the sequence `pushv x; pop x` from a no-op.

3 Types and Assertions

As well as divergence and normal termination, programs can get stuck as a result of type errors applying basic operations, accessing undefined variables, underflowing either of the stacks, or jumping or calling outside the program. We rule out such behaviour using a type system, and define assertions relative to those types. This seems natural, but it is not the only reasonable way to proceed – although both the JVM and CLR have

$$\Theta; \Delta; \Sigma \vdash n : \texttt{int} \qquad \Theta; \Delta; \Sigma \vdash b : \texttt{bool}$$

$$\Theta; \Delta, x : \tau; \Sigma \vdash x : \tau \qquad \Theta, a : \tau; \Delta; \Sigma \vdash a : \tau$$

$$\Theta; \Delta; \Sigma, \tau \vdash s(0) : \tau \qquad \frac{\Theta; \Delta; \Sigma \vdash s(i) : \tau}{\Theta; \Delta; \Sigma, \tau' \vdash s(i+1) : \tau}$$

$$\frac{\Theta, a : \tau; \Delta; \Sigma \vdash E : \texttt{bool}}{\Theta; \Delta; \Sigma \vdash \forall a \in \tau.E : \texttt{bool}} \quad \frac{\Theta; \Delta; \Sigma \vdash E : \tau_1 \quad \mathit{uop} : \tau_1 \to \tau_2}{\Theta; \Delta; \Sigma \vdash \mathit{uop}\ E : \tau_2}$$

$$\frac{\Theta; \Delta; \Sigma \vdash E_1 : \tau_1 \quad \Theta; \Delta; \Sigma \vdash E_2 : \tau_2 \quad \mathit{bop} : \tau_1 \times \tau_2 \to \tau_3}{\Theta; \Delta; \Sigma \vdash E_1\ \mathit{bop}\ E_2 : \tau_3}$$

Fig. 2. Expression Typing

type-checkers ('verifiers'), the CLR does give a semantics to unverifiable code, which can be executed if it has been granted sufficient permissions. Our type-based approach prevents one from proving any properties at all of unverifiable code.

3.1 Basic Types and Expressions

A *base type*, τ, is either `int` or `bool`. A *stack type*, Σ, is a finite sequence $\tau_1, \ldots, \tau_n$ of base types. A *store type*, Δ, is a finite map $x_1 : \tau_1, \ldots, x_n : \tau_n$ from program variables to base types. We assume a set of *auxiliary variables*, ranged over by a. An *auxiliary variable context*, Θ, is a finite map from auxiliary variables to base types. A *valuation*, ρ is a function from auxiliary variables to values. We write $\rho : a_1 : \tau_1, \ldots, a_n : \tau_n$ for $\forall 1 \leq i \leq n.\ \rho(a_i) : \tau_i$.

Our low-level machine does not deal directly with complex expressions, but we will use them in forming assertions. Their grammar is as follows:

$$E \ := \ \underline{n} \mid \underline{b} \mid x \mid a \mid s(i) \mid E\ \mathit{bop}\ E \mid \mathit{uop}\ E \mid \forall a \in \tau.E$$

The expression form $s(i)$, for i a natural number, represents the ith element down the stack. Note that universal quantification over integers and booleans is an expression form. We assume that at least equality and a classical basis set of propositional connectives (e.g. negation and conjunction) are already operators in the language; otherwise we would simply add them to expressions. In any case, we will feel free to use fairly arbitrary first-order arithmetic formulae (including existential quantification and inductively defined predicates) in assertions, regarding them as syntactic sugar for, or standard extensions of, the above grammar.

Expressions are assigned base types in the context of a given stack typing, store typing and auxiliary variable context by the rules shown in Figure 2. Expression typing satisfies the usual weakening properties, and the definitions of, and typing lemmas concerning, substitutions $E[E'/x]$, $E[E'/a]$ and $E[E'/s(i)]$ are as one would expect.

$$\llbracket \underline{v} \rrbracket\,\rho\,G\,\sigma \;=\; v \qquad \llbracket a \rrbracket \rho\,G\,\sigma \;=\; \rho(a) \qquad \llbracket x \rrbracket\,\rho\,G\,\sigma \;=\; G(x)$$
$$\llbracket s(0) \rrbracket\,\rho\,G\,(\sigma,v) \;=\; v \qquad \llbracket s(i+1) \rrbracket\,\rho\,G\,(\sigma,v) \;=\; \llbracket s(i) \rrbracket\,\rho\,G\,\sigma$$
$$\llbracket \underline{uop}\,E \rrbracket\,\rho\,G\,\sigma \;=\; uop\,(\llbracket E \rrbracket\,\rho\,G\,\sigma)$$
$$\llbracket E_1\,\underline{bop}\,E_2 \rrbracket\,\rho\,G\,\sigma \;=\; (\llbracket E_1 \rrbracket\,\rho\,G\,\sigma)\;bop\;(\llbracket E_2 \rrbracket\,\rho\,G\,\sigma)$$
$$\llbracket \forall a \in \tau.E \rrbracket\,\rho\,G\,\sigma \;=\; \bigwedge_{v \in \llbracket \tau \rrbracket} \llbracket E \rrbracket\,\rho[a \mapsto v]\,G\,\sigma$$

Fig. 3. Expression Semantics

If $\Theta; \Delta; \Sigma \vdash E : \tau$, $\rho : \Theta$, $G : \Delta$ and $\sigma : \Sigma$ then we define $\llbracket E \rrbracket\,\rho\,G\,\sigma \in \llbracket \tau \rrbracket$ as in Figure 3. The semantics is well defined and commutes with each of our three forms of substitution. If $\Theta; \Delta; \Sigma \vdash E_i : \texttt{bool}$ for $i \in \{1,2\}$, we write $\Theta; \Sigma; \Delta \models E_1 \implies E_2$ to mean

$$\forall \rho : \Theta. \forall G : \Delta. \forall \sigma : \Sigma.\; \llbracket E_1 \rrbracket\,\rho\,G\,\sigma \implies \llbracket E_2 \rrbracket\,\rho\,G\,\sigma.$$

where $\implies$ is classical first-order implication. We define syntactic operations $\mathit{shift}(E)$ and $E \setminus\!\setminus E'$ for reindexing expressions when the stack is pushed or popped by

E	$\mathit{shift}(E)$	$E \setminus\!\setminus E'$
$s(0)$	$s(1)$	E'
$s(i+1)$	$s(i+2)$	$s(i)$
$E_1\ bop\ E_2$	$\mathit{shift}(E_1)\ bop\ \mathit{shift}(E_2)$	$(E_1 \setminus\!\setminus E')\ bop\ (E_2 \setminus\!\setminus E')$
$uop\ E_1$	$uop\ (\mathit{shift}(E_1))$	$uop\ (E_1 \setminus\!\setminus E')$
$\forall a \in \tau.E_1$	$\forall a \in \tau.\mathit{shift}(E_1)$	$\forall a \in \tau.(E_1 \setminus\!\setminus E')$ capture-avoiding
otherwise	E	E

where the $E \setminus\!\setminus E'$ operation, combining substitution for $s(0)$ with 'unshifting', is defined when $\Theta; \Delta; \Sigma, \tau' \vdash E : \tau$ and $\Theta; \Delta; \Sigma \vdash E' : \tau'$.

3.2 Types and Assertions for Programs

One could first present a type system and then a second inference system for assertion checking. Since the structure of the two inference systems would be similar, and we need types in defining assertions, we instead combine both into one system. The type part used here is monomorphic and somewhat restrictive, rather like that of the JVM. Over this we layer a richer assertion language, including explicit universal quantification. We define the structure of, and axiomatise entailment on, this assertion language explicitly (rather than delegating both to some ambient higher-order logic).

An *extended label type*, χ, is a universally-quantified pair of a precondition and a postcondition, where the pre- and postconditions each comprise a store type, a stack type and a boolean-valued expression:

$$\chi \;:=\; \Delta; \Sigma; E \to \Delta'; \Sigma'; E' \mid \forall a : \tau.\chi$$

A *label environment* is a finite mapping from labels to extended label types

$$\Gamma \;:=\; l_1 : \chi_1, \ldots, l_n : \chi_n$$

Order:

$$\frac{\Theta \vdash \chi \ \texttt{ok} \quad \Theta \vdash \hat{E} : \texttt{bool}}{\Theta; \hat{E} \vdash \chi \leq \chi}\ \text{refl} \qquad \frac{\Theta; \hat{E} \vdash \chi \leq \chi' \quad \Theta; \hat{E} \vdash \chi' \leq \chi''}{\Theta; \hat{E} \vdash \chi \leq \chi''}\ \text{trans}$$

Quantifier:

$$\frac{\Theta \vdash \forall a : \tau.\chi \ \texttt{ok} \quad \Theta \vdash E : \tau \quad \Theta \vdash \hat{E} : \texttt{bool}}{\Theta; \hat{E} \vdash \forall a : \tau.\chi \leq \chi[E/a]}\ \forall\text{-subs}$$

$$\frac{\Theta \vdash \chi' \ \texttt{ok} \quad \Theta \vdash \hat{E} : \texttt{bool} \quad \Theta, a : \tau; \hat{E} \vdash \chi' \leq \chi}{\Theta; \hat{E} \vdash \chi' \leq \forall a : \tau.\chi}\ \forall\text{-glb}$$

Arrow:

$$\frac{\Theta; \Delta; \Sigma \vdash F \wedge \hat{E} \implies E \qquad \Theta; \Delta'; \Sigma' \vdash E' \wedge \hat{E} \implies F'}{\Theta; \hat{E} \vdash (\Delta; \Sigma; E \to \Delta'; \Sigma'; E') \leq (\Delta; \Sigma; F \to \Delta'; \Sigma'; F')}\ \to$$

$$\frac{\Theta \vdash \hat{E} : \texttt{bool} \quad \Theta, a : \tau; \Delta; \Sigma \vdash E : \texttt{bool} \quad \Theta; \Delta'; \Sigma' \vdash E' : \texttt{bool}}{\Theta; \hat{E} \vdash \forall a : \tau.(\Delta; \Sigma; E \to \Delta'; \Sigma', E') \leq (\Delta; \Sigma; \exists a \in \tau.E \to \Delta'; \Sigma'; E')}\ \forall\exists \to$$

Frame:

$$\frac{\Theta \vdash \hat{E} : \texttt{bool} \quad \Theta; \overline{\Delta}; \overline{\Sigma} \vdash I : \texttt{bool} \quad \Theta \vdash \Delta; \Sigma; E \to \Delta'; \Sigma'; E' \ \texttt{ok}}{\begin{array}{l}\Theta; \hat{E} \vdash \ (\Delta; \Sigma; E \to \Delta'; \Sigma'; E') \\ \qquad \leq (\overline{\Delta}, \Delta; \overline{\Sigma}, \Sigma; \mathit{shift}^{|\Sigma|}(I) \wedge E \to \overline{\Delta}, \Delta'; \overline{\Sigma}, \Sigma'; \mathit{shift}^{|\Sigma'|}(I) \wedge E')\end{array}}$$

Fig. 4. Subtyping/Entailment for Extended Label Types

These are subject to the following well-formedness conditions:

$$\frac{\Theta \vdash \chi_1 \ \texttt{ok} \quad \cdots \quad \Theta \vdash \chi_n \ \texttt{ok}}{\Theta \vdash l_1 : \chi_1, \ldots, l_n : \chi_n \ \texttt{ok}} \qquad \frac{\Theta, a : \tau \vdash \chi \ \texttt{ok}}{\Theta \vdash \forall a : \tau.\chi \ \texttt{ok}}$$

$$\frac{\Theta; \Delta; \Sigma \vdash E : \texttt{bool} \quad \Theta; \Delta'; \Sigma' \vdash E' : \texttt{bool}}{\Theta \vdash \Delta; \Sigma; E \to \Delta'; \Sigma'; E' \ \texttt{ok}}$$

The intuitive meaning of $l : \Delta; \Sigma; E \to \Delta'; \Sigma'; E'$ is that if one jumps to l with a store of type Δ and a stack of type Σ, such that E is true, then the program will, without getting stuck, either diverge, `halt`, or reach a `ret` with the callstack unchanged and a store of type Δ' and a stack of type Σ' such that E' is true. We will formalise (a more extensional version of) this intuition in Section 4.

We define $\chi[E/a]$ in the obvious capture-avoiding way and axiomatise entailment on well-formed extended label types as shown in Figure 4. The basic entailment judgement has the form $\Theta; \hat{E} \vdash \chi \leq \chi'$ where $\Theta \vdash \hat{E} : \texttt{bool}$ (we elide store and stack types here), $\Theta \vdash \chi \ \texttt{ok}$ and $\Theta \vdash \chi' \ \texttt{ok}$. The purpose of $\hat{E}$, which will also show up in the rules of the program logic proper, is to constrain the values taken by the variables in

Θ. Including $\hat{E}$ in judgements does not seem necessary for proving properties of closed programs, but we shall see later how it helps us to reason in a modular fashion about program fragments.

The $[\rightarrow]$ rule is basically the usual one for subtyping function types, here playing the role of Hoare logic's rule of consequence. The $[\forall\exists \rightarrow]$ rule is a kind of internalization of the usual left rule for existential quantification. Note how in these two rules, classical first-order logic, which we do not analyse further, interacts with our more explicit, and inherently intuitionisitic, program logic.

The most complex and interesting case is the frame rule, which is closely related to the rule of the same name in separation logic [25].[1] This allows an invariant I to be added to the pre and postconditions of an extended label type χ, provided that invariant depends only on store and stack locations that are guaranteed to be disjoint from the footprint of the program up to a return to the current top of the callstack. Note how references to stack locations in the invariant are adapted by shifting. The frame rule allows assumptions about procedures to be locally adapted to each call site, which is necessary for modular reasoning. Rather than a single separating conjunction $*$ on assertions, we use our 'tight' (multiplicative) interpretation of state types to ensure separation and use ordinary (additive) conjunction on the assertions themselves.

In use, of course, one needs to adapt extended types to contexts in which there is some relationship between the variables and stack locations mentioned in Δ and Σ and those added in $\overline{\Delta}$ and $\overline{\Sigma}$. This is achieved by using (possibly new) auxiliary variables to split the state dependency before applying the frame rule: see Example 4 in Section 5 for a simple example.

3.3 Assigning Extended Types to Programs

Our basic judgement form is $\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma'$ where Γ and Γ' are label environments with disjoint domains, $\hat{E}$ is a boolean-valued expression and p is a program fragment.

The context Γ expresses assumptions about imported code that will be subsequently linked with p, whilst Γ' says what p will guarantee about exported labels, given those assumptions. Thus none of the labels in Γ, and all of the labels in Γ', will be in the domain of p. The rules for assigning extended types to programs are shown (eliding some obvious well-formedness conditions in a vain attempt to improve readibility) in Figures 5 and 6.

The key structural rule is [link], which allows proved program fragments to be concatenated. The rule has a suspiciously circular nature: if p_1 guarantees Γ_1 under assumptions Γ_2, and p_2 guarantees Γ_2 under assumptions Γ_1, then p_1 linked with p_2 guarantees both Γ_1 and Γ_2 unconditionally. The rule *is*, however, sound for our partial correctness (safety) interpretation, as we prove later.

The $[\forall\text{-r}]$ rule is a mild variant of the usual introduction/right rule for universal quantification. The auxiliary variable a does not appear free in Γ, so we may universally quantify it in each (hence the vector notation) of the implictly conjoined conclu-

[1] Since our rule concerns both 'frame properties' and 'frames' in the sense of activation records, it arguably has even more claim on the name :-).

$$\frac{\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma', l : \chi}{\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma'}\ \text{widthr}$$

$$\frac{\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma' \quad \Theta \vdash \chi\ \texttt{ok} \quad l \notin dom(p)}{\Theta; \hat{E}; \Gamma, l : \chi \vdash p \rhd \Gamma'}\ \text{widthl}$$

$$\frac{\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma', l : \chi \quad \Theta; \hat{E} \vdash \chi \leq \chi'}{\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma', l : \chi'}\ \text{subr}$$

$$\frac{\Theta; \hat{E}; \Gamma, l : \chi \vdash p \rhd \Gamma' \quad \Theta; \hat{E} \vdash \chi' \leq \chi}{\Theta; \hat{E}; \Gamma, l : \chi' \vdash p \rhd \Gamma'}\ \text{subl}$$

$$\frac{\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma' \quad \Theta, \Theta' \vdash \hat{E}' \implies \hat{E}}{\Theta, \Theta'; \hat{E}'; \Gamma \vdash p \rhd \Gamma'}\ \text{ctxl}$$

$$\frac{\Theta; \hat{E}; \Gamma \vdash p \rhd \boldsymbol{l_i} : \boldsymbol{\Delta_i}; \boldsymbol{\Sigma_i}; \boldsymbol{E_i} \to \boldsymbol{\Delta'_i}; \boldsymbol{\Sigma'_i}; \boldsymbol{E'_i}}{\Theta; true; \Gamma \vdash p \rhd \boldsymbol{l_i} : \boldsymbol{\Delta_i}; \boldsymbol{\Sigma_i}; \hat{E} \wedge \boldsymbol{E_i} \to \boldsymbol{\Delta'_i}; \boldsymbol{\Sigma'_i}; \boldsymbol{E'_i}}\ \text{ctxr}$$

$$\frac{\Theta \vdash \Gamma\ \texttt{ok} \quad \Theta \vdash \hat{E} : \texttt{bool} \quad \Theta, a : \tau; \hat{E}; \Gamma \vdash p \rhd \boldsymbol{l_i} : \boldsymbol{\chi_i}}{\Theta; \hat{E}; \Gamma \vdash p \rhd \boldsymbol{l_i} : \forall a : \tau . \boldsymbol{\chi_i}}\ \forall\text{-r}$$

$$\frac{\Theta; \hat{E}; \Gamma, \Gamma_2 \vdash p_1 \rhd \Gamma_1 \quad \Theta; \hat{E}; \Gamma, \Gamma_1 \vdash p_2 \rhd \Gamma_2}{\Theta; \hat{E}; \Gamma \vdash p_1, p_2 \rhd \Gamma_1, \Gamma_2}\ \text{link}$$

Fig. 5. Program Logic: Structural and Logical Rules

sions. The vector notation also appears in the [ctxr] rule, allowing global conditions on auxiliary variables to be transferred to the preconditions of each of the conclusions. [2]

The reader will notice that the axioms fail to cope with branch or call instructions whose target is the instruction itself, as we have said that judgements in which the same label appears on the left and right are ill-formed. This is easily rectified either by adding special case rules, or[3] by a more general relaxation of our requirement for imports Γ and exports Γ' to be disjoint, but we omit the (uncomplicated) details here. We also remark that if we are willing to make aggressive use of the frame rule, subtyping and auxiliary variable manipulations, the axioms can be presented in a more stripped-down form. For example, the rule for `ret` can be presented as just

$$-; true; - \vdash [l : \texttt{ret}] \rhd l : -; -; true \to -; -; true$$

[2] Equivalently, one could state these two rules with a single conclusion and add a right rule for conjunction. Our presentation threads the subject program p linearly through the derivation, making it clear that we only analyse its internal structure once.

[3] Thanks to one of the referees for this observation.

$$\Theta; \hat{E}; \Gamma \vdash [l : \texttt{halt}] \rhd l : \chi$$

$$\begin{array}{l}\Theta; \hat{E}; \Gamma, l+1 : \Delta; \Sigma, \tau; E \to \Delta'; \Sigma'; E' \\ \vdash [l : \texttt{pushc}\, v] \rhd l : \Delta; \Sigma; E \setminus\!\setminus v \to \Delta'; \Sigma'; E'\end{array} \text{(where } v : \tau)$$

$$\begin{array}{l}\Theta; \hat{E}; \Gamma, l+1 : \Delta, x : \tau; \Sigma, \tau; E \to \Delta'; \Sigma'; E' \\ \vdash [l : \texttt{pushv}\, \texttt{x}] \rhd l : \Delta, x : \tau; \Sigma; E \setminus\!\setminus x \to \Delta'; \Sigma'; E'\end{array}$$

$$\begin{array}{l}\Theta; \hat{E}; \Gamma, l+1 : \Delta, x : \tau; \Sigma; E \to \Delta'; \Sigma'; E' \\ \vdash [l : \texttt{pop}\, \texttt{x}] \rhd l : \Delta, x : \tau'; \Sigma, \tau; \mathit{shift}(E)[s(0)/x] \to \Delta'; \Sigma'; E'\end{array}$$

$$\Theta; \hat{E}; \Gamma, l+1 : \Delta; \Sigma, \tau, \tau; E \to \Delta'; \Sigma'; E' \vdash [l : \texttt{dup}] \rhd l : \Delta; \Sigma, \tau; E \setminus\!\setminus s(0) \to \Delta'; \Sigma'; E'$$

$$\begin{array}{l}\Theta; \hat{E}; \Gamma, l+1 : \Delta; \Sigma, \tau_3; E \to \Delta'; \Sigma'; E' \\ \vdash [l : \texttt{binop}_{bop}] \rhd l : \Delta; \Sigma, \tau_1, \tau_2; \mathit{shift}(E)[(s(1)\, \mathit{bop}\, s(0))/s(1)] \to \Delta'; \Sigma'; E' \\ \qquad \text{(where } \mathit{bop} : \tau_1 \times \tau_2 \to \tau_3)\end{array}$$

$$\begin{array}{l}\Theta; \hat{E}; \Gamma, l+1 : \Delta; \Sigma, \tau_2; E \to \Delta'; \Sigma'; E' \\ \vdash [l : \texttt{unop}_{uop}] \rhd l : \Delta; \Sigma, \tau_1; E[(\mathit{uop}\, s(0))/s(0)] \to \Delta'; \Sigma'; E'\end{array} (\mathit{uop} : \tau_1 \to \tau_2)$$

$$\begin{array}{l}\Theta; \hat{E}; \Gamma, l+1 : \Delta; \Sigma; E \setminus\!\setminus \mathit{false} \to \Delta'; \Sigma'; E' ,\, l' : \Delta; \Sigma; E \setminus\!\setminus \mathit{true} \to \Delta'; \Sigma'; E' \\ \vdash [l : \texttt{brtrue}\, l'] \rhd l : \Delta; \Sigma, \texttt{bool}; E \to \Delta'; \Sigma'; E'\end{array}$$

$$\begin{array}{l}\Theta; \hat{E}; \Gamma, l+1 : \Delta''; \Sigma''; E'' \to \Delta'; \Sigma'; E' ,\, l' : \Delta; \Sigma; E \to \Delta''; \Sigma''; E'' \\ \vdash [l : \texttt{call}\, l'] \rhd l : \Delta; \Sigma; E \to \Delta'; \Sigma'; E'\end{array}$$

$$\Theta; \hat{E}; \Gamma \vdash [l : \texttt{ret}] \rhd l : \Delta; \Sigma; E \to \Delta; \Sigma; E$$

Fig. 6. Program Logic: Instruction-Specific Axioms

4 Semantics of Types and Assertions

One could certainly formulate and prove a correctness theorem for this logic syntactically, using a ‘preservation and progress’ argument. Technically, the syntactic approach is probably the simplest way of proving soundness, though it has the mild disadvantage of requiring a somewhat artifical extension of the typing and logical rules to whole configurations, rather than just the programs with which one starts. More fundamentally, the syntactic approach fails to capture the *meaning* of types and assertions, which, although this is partly a question of taste, I believe to be more than a philosophical objection.

In practice, we would like to be able safely to link low-level components that have been verified using different proof systems and would arguably also like to have a formal statement of the invariants that *should* be satisfied by trusted-but-unverified components. These goals require a notion of semantics for types and assertions that is formulated in terms of the observable behaviour of programs, independent of a particular syntactic inference system. A syntactic approach to the semantics of program logics can also be excessively intensional, distinguishing observationally equivalent programs in a way that may weaken the logic for applications such as program transformation. Since we are making no claims here about completeness of our logic, we refrain from pushing this argument more strongly, however.

The way in which we choose to formulate an extensional semantics for types and assertions is via the notion of orthogonality with respect to contexts ('perping'). This is a general pattern, related to continuation-passing and linear negation, that has been applied in a number of different operational settings in recent years, including structuring semantics, defining operational versions of admissible predicates, logical relations [27] and ideal models for types [35], and proving strong normalization [19].

To establish the soundness of our link rule we also find it convenient to index our semantic definitions by step-counts, a technique that Appel and his collaborators have used extensively in defining semantic interpretations of types over low-level languages [3, 4, 2]. By contrast with our use of orthogonality, which is a deliberate choice of what we regard as the 'right' semantics, the use of indexing is essentially a technical device to make the operational proofs go through.

Assume $\Theta; \Delta; \Sigma \vdash E : \texttt{bool}$ and $\rho : \Theta$. We define

$$\mathbb{E}_\rho(\Delta; \Sigma; E) \subseteq \mathit{Stores} \times \mathit{Stacks} \stackrel{def}{=} \{(G, \sigma) \mid G : \Delta \wedge \sigma : \Sigma \wedge [\![E]\!]\, \rho\, G\, \sigma = \mathit{true}\}$$

If $S \subseteq \mathit{Stores} \times \mathit{Stacks}$ and $k \in \mathbb{N}$, we define

$$S_k^\top \subseteq \mathit{Configs} = \{\langle p|C|G'|\sigma'|l\rangle \mid \forall (G, \sigma) \in S. \mathit{Safe}_k \langle p|C|G', G|\sigma', \sigma|l\rangle\}$$

So $S_k^\top$ is the set of configurations that, when extended with any state in S, are safe for k steps: think of these as (k-approximate) 'test contexts' for membership of S.

Now for $\Theta \vdash \Gamma\ \texttt{ok}$, $\rho : \Theta$ and $k \in \mathbb{N}$, define $\models_\rho^k p \rhd \Gamma$ inductively as follows:

$$\models_\rho^k p \rhd l_1 : \chi_1, ..., l_n : \chi_n \iff \textstyle\bigwedge_{i=1}^n . \models_\rho^k p \rhd l_i : \chi_i$$

$$\models_\rho^k p \rhd l : \forall a \in \tau . \chi \iff \forall x \in [\![\tau]\!]. \models_{\rho[a \mapsto x]}^k p \rhd l : \chi$$

$$\begin{aligned}\models_\rho^k p \rhd l : \Delta; \Sigma; E \to \Delta'; \Sigma'; E' \iff & \forall (G, \sigma) \in \mathbb{E}_\rho(\Delta; \Sigma; E). \\ & \forall \langle p, p'|C|G'|\sigma'|l'\rangle \in \mathbb{E}_\rho(\Delta'; \Sigma'; E')_k^\top . \mathit{Safe}_k \langle p, p'|C, l'|G', G|\sigma', \sigma|l\rangle\end{aligned}$$

The important case is the last one: a program p satisfies $l : \Delta; \Sigma; E \to \Delta'; \Sigma'; E'$ to a k-th approximation if for any k-test context for E' that extends p and has entry point l', if one pushes l' onto the call stack, extends the state with one satisfying E, and commences execution at l, then the overall result is safe for k steps.[4]

We then define the semantics of contextual judgements by

$$\begin{aligned}&\Theta; \hat{E}; \Gamma \models p \rhd \Gamma' \\ \iff & \forall \rho : \Theta . \forall k \in \mathbb{N}. \forall p'. \; [\![\hat{E}]\!]\rho = \mathit{true} \wedge \models_\rho^k p', p \rhd \Gamma \implies \models_\rho^{k+1} p', p \rhd \Gamma'\end{aligned}$$

[4] It would actually suffice only to ask the context to be safe for $k - 1$ steps, rather than k.

So p satisfies Γ' under assumptions Γ if, for all k, any extension of p that satisfies Γ for k steps satisfies Γ' for $k+1$ steps. The following theorem establishes the semantic soundness of the entailment relation on extended label types, and is proved by induction on the rules in Figure 4:

Theorem 1. *If $\Theta; \hat{E} \vdash \chi \leq \chi'$ then for all p, l, $\rho : \Theta$, $k \in \mathbb{N}$*

$$[\![\hat{E}]\!]\rho = \mathit{true} \wedge \models^k_\rho p \rhd l : \chi \implies \models^k_\rho p \rhd l : \chi'.$$

We then use Theorem 1 and a further induction on the rules in Figures 5 and 6 to establish the semantic soundness of the program logic:

Theorem 2. *If $\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma'$ then $\Theta; \hat{E}; \Gamma \models p \rhd \Gamma'$.*

5 Examples

Our logic is very fine-grained (and the judgement forms fussily baroque), so proofs of any non-trivial example programs are lengthy and extremely tedious to construct by hand. In this section we just present a few micro-examples, demonstrating particular features of the logic. We hope these convince the reader that, given sufficient patience, one can indeed prove all the program properties one might expect (subject to the limitations of the simple type system, of course), and do so in a fairly arbitrarily modular fashion. The technical report contains more details of these examples, as well a simple example of mutual recursion.

Example 1. It takes around twelve detailed steps to derive

$$-; \mathit{true}; - \vdash [0 : \texttt{pushc}\ 1,\ 1 : \texttt{binop}_+,\ 2 : \texttt{ret}] \rhd 0 : \chi_0 \tag{1}$$

where

$$\chi_0 = \forall a : \texttt{int}.(-; \texttt{int}; s(0) = a \to -; \texttt{int}; s(0) = a + 1)$$

which establishes that for any integer a, if we call label 0 with a on the top of the stack, the fragment will either `halt`, diverge or `return` with $a + 1$ on the top of the stack.

Example 2. Now consider the following simple fragment:

$$[10 : \texttt{call}\ 0, 11 : \texttt{br}\ 0]$$

which one may think of as a tail-call optimized client of the code in the first example. Write χ_0' for

$$\forall c : \texttt{int}.(-; \texttt{int}; (s(0) = c) \wedge ((c = b) \vee (c = b + 1)) \to -; \texttt{int}; s(0) = c + 1)$$

which is well-formed in the context $b : \texttt{int}$. It takes seven steps to show

$$\begin{aligned} &b : \texttt{int}; \mathit{true}; 0 : \chi_0' \vdash \\ &\ [10 : \texttt{call}\ 0, 11 : \texttt{br}\ 0] \rhd 10 : -; \texttt{int}; s(0) = b \to -; \texttt{int}; s(0) = b + 2 \end{aligned} \tag{2}$$

which establishes (roughly) that for any b, if the code at label 0 can be relied upon to compute the successors of b and of $b+1$, then the code at label 10 guarantees to compute $b+2$. We now consider linking in the code from the first example. Another eight or so steps of reasoning with entailment and structural rules allows us to combine judgements (1) and (2) to obtain

$$\begin{aligned}&-; \mathit{true}; - \vdash [0 : \mathtt{pushc}\,1,\ 1 : \mathtt{binop}_+,\ 2 : \mathtt{ret},\ 10 : \mathtt{call}\,0, 11 : \mathtt{br}\,0] \rhd \\ &0 : \chi_0,\ 10 : \forall b : \mathtt{int}.(-; \mathtt{int}; s(0) = b \to -; \mathtt{int}; s(0) = b+2)\end{aligned}$$

establishing that for any integer b, calling the code at label 10 with b returns with $b+2$. The point about this example is to demonstrate a certain style of modular reasoning: the proof about the code at 10 and 11 was carried out under a rather weak assumption about the code at 0. *After* linking the two fragments together, we were able to generalize and conclude a stronger result about the code at 10 in the composed program *without* re-analysing either code fragment. To re-emphasize this point, we now consider replacing the code at 0 with something weaker.

Example 3. Given the source program

$$\begin{aligned}p = \ &[0 : \mathtt{dup},\ 1 : \mathtt{pushc}\,7,\ 2 : \mathtt{binop}_<,\ 3 : \mathtt{brtrue}\,5,\ 4 : \mathtt{ret}, \\ &5 : \mathtt{pushc}\,1,\ 6 : \mathtt{binop}_+,\ 7 : \mathtt{ret}]\end{aligned}$$

we can prove, using the rule for conditional branches, that

$$-; \mathit{true}; - \vdash p \rhd 0 : \forall a : \mathtt{int}.(-; \mathtt{int}; a < 7 \land s(0) = a \to -; \mathtt{int}; s(0) = a+1) \tag{3}$$

showing that the code at label 0 computes the successor of all integers smaller than 7.

We now consider [link]ing the judgement (3) with that we derived for the client program (2). With a few purely logical manipulations (using $\hat{E} = b < 6$) we can derive

$$\begin{aligned}&-; \mathit{true}; - \vdash p, [10 : \mathtt{call}\,0, 11 : \mathtt{br}\,0] \\ &\rhd 10 : \forall b : \mathit{int}.(-; \mathtt{int}; b < 6 \land s(0) = b \to -; \mathtt{int}; s(0) = b+2)\end{aligned}$$

showing that calling 10 now computes $b+2$ for all b less than 6. Again, we did not reanalyse the client code, but were able to propogate the information about the range over which our 'partial successor' code at 0 works through the combined program *after* linking. The inclusion of $\hat{E}$, or something equivalent, seems necessary for this kind of reasoning: we need to add constraints on auxiliary variables throughout a judgement, as well as to assumptions or conclusions about individual labels.

Example 4. As a simple example of how our entailment relation allows extended types for labels to be adapted for particular calling contexts, consider the assertion χ_0 we had in our first example. Eight small steps of reasoning with the entailment rules allow one to deduce

$$-; \mathit{true} \vdash \chi_0 \leq (-; \mathtt{int}, \mathtt{int}; (s(1) < s(0) \to -; \mathtt{int}, \mathtt{int}; s(1) < s(0))$$

So, although χ_0 only mentions a one-element stack, when one calls a label assumed to satisfy χ_0 one can locally adapt that assumption to the situation where are two things on the stack *and* a non-trivial relationship between them. This is a common pattern: we use the frame rule and new auxiliary variables to add a separated invariant and then existentially quantify the new variables away.

Example 5. Consider the source procedure

```
void f() {
  x := 0;
  while(x<5) {
    x := x+1;
  }
}
```

A typical Java or $C^\sharp$ compiler will compile the loop with the test and conditional backwards branch at the end, preceded by a header which branches unconditionally into the loop to execute the test the first time. This yields code p something like

$$\begin{array}{l}[1 : \texttt{pushc}\, 0,\ 2 : \texttt{pop}\, \texttt{x},\ 3 : \texttt{pushc}\, \mathit{true},\ 4 : \texttt{brtrue}\, 9,\ 5 : \texttt{pushv}\, \texttt{x},\\ \ 6 : \texttt{pushc}\, 1,\ 7 : \texttt{binop}_+,\ 8 : \texttt{pop}\, \texttt{x},\ 9 : \texttt{pushv}\, \texttt{x},\ 10 : \texttt{pushc}\, 5,\ 11 : \texttt{binop}_<,\\ \ 12 : \texttt{brtrue}\, 5,\ 13 : \texttt{ret}]\end{array}$$

Such unstructured control-flow makes no difference to reasoning in our low-level logic: as one would hope, we can easily derive

$$-; \mathit{true}; - \vdash p \rhd 0 : x : \texttt{int}; -; \mathit{true} \to x : \texttt{int}; -; x = 5.$$

Example 6. Although our machine has call and return instructions, it does not specify any particular calling convention or even delimit entry points of procedures. Both the machine and the logic can deal with differing calling conventions and multiple entry points. For example, given

$$p \;=\; [1 : \texttt{pushv}\, \texttt{x},\ 2 : \texttt{pushc}\, 1,\ 3 : \texttt{binop}_+,\ 4 : \texttt{ret}]$$

we can derive

$$\begin{array}{l}-; \mathit{true}; - \vdash p \rhd\\ \quad 1 : \forall a : \texttt{int}.(x : \texttt{int}; -; x = a \to x : \texttt{int}; \texttt{int}; x = a \wedge s(0) = a + 1),\\ \quad 2 : \forall a : \texttt{int}.(-; \texttt{int}; s(0) = a \to -; \texttt{int}; s(0) = a + 1)\end{array}$$

so one can either pass a parameter in the variable x, calling address 1, or on the top of the stack, calling address 2.

6 Discussion

We have presented a typed program logic for a simple stack-based intermediate language, bearing roughly the same relationship to Java bytecode or CIL that a language of while-programs with procedures does to Java or $C^\sharp$.

The contributions of this work include the modular treatment of program fragments and linking (similar to, for example, [11]); the explicit treatment of different kinds of contexts and quantification; the interplay between the prescriptive, tight interpretation of types and the descriptive interpretation of expressions, leading to a separation-logic style treatment of adaptation; the use of shifting to reindex assertions; dealing with non-trivially unstructured control flow (including multiple entry points to mutually-recursive procedures) and an indexed semantic model based on perping.

There is some related work on logics for bytecode programs. Borgström [10] has approached the problem of proving bytecode programs meet specifications by first decompiling them into higher-level structured code and then reasoning in standard Floyd-Hoare logic. Quigley [29, 30] has formalized rules for Hoare-like reasoning about a small subset of Java bytecode within Isabelle, but her treatment is based on trying to rediscover high-level control structures (such as while loops); this leads to rules which are both complex and rather weak. More recently, Bannwart and Müller [6] have combined the simple logic of an early draft of the present paper [7] with a higher-level, more traditional Hoare logic for Java to obtain a rather different logic for bytecodes than that we present here. We should also mention the work of Aspinall et al on a VDM-like logic for resource verification of a JVM-like language [5].

Even for high-level languages, satisfactory accounts of auxiliary variables and rules for adaptation in Hoare logics for languages with procedures seem to be surprisingly recent, see for example the work of Kleymann [18] and von Oheimb & Nipkow [34, 26]. Our fussiness about contexts and quantification, and use of substructural ideas, differs from most of this other work, leading to a rather elegant account of invariants of procedures and a complete absence of side-conditions. The use of auxiliary variables scoped across an entire judgement, and explicit universal quantification (rather than implicit, closing, quantification on each triple in the context) seems much the best way to reason compositionally, allowing one to relate assumptions on different labels, but has previously been shied away from. As von Oheimb [34] says

> A real solution would be explicit quantification like $\forall Z.\{P\,Z\}\,c\,\{Q\,Z\}$, but this changes the structure of Hoare triples and makes them more difficult to handle. Instead we prefer implicit quantification at the level of triple validity[...]

The other line of closely related research is on proof-carrying code [24, 23] and typed assembly languages [22], much of which has a similar 'logical' flavour to this, with substructural ideas having been applied to stacks [17], heaps [20] and aliasing [32]. Our RISC-like stack-based low-level machine, with no built-in notion of procedure entry points or calling conventions, is similar to that of STAL [21]. Compared with most of the cited work, we have a much simpler machine (no pointer manipulation, dynamic allocation or code pointers), but go beyond simple syntactic type soundness to give a richer program logic with a semantic interpretation. Especially close is the work of Appel et al on semantic models of types in foundational proof-carrying code, from which we borrowed the step-indexed proof technique, and of Shao and Hamid [14] on interfacing Hoare logic with a syntactic type system for low-level code as a way of verifying linkage between typed assembly language modules verified using different systems.

One might (and all the referees did) reasonably ask why we have not followed most of the recent work in this area by, firstly, formalizing our logic in a theorem prover and, secondly, avoiding all the explicit treatment of quantification, auxiliary variables etc. in favour of inheriting them from a shallow embedding of the semantics in an ambient higher-order logic. Machine checking is certainly helpful in avoiding unsoundness, is probably essential for managing all the details of logics for realistic-scale languages or machines, and is a necessary component in PCC-like deployment scenarios. We certainly plan to investigate mechanization in the near future, but believe a traditional pencil-and-paper appraoch is quite reasonable for preliminary investigations with toy calculi, despite the history of unsound Hoare logic rules in the literature. We certainly made errors in earlier versions of this paper but, given an independent semantics and formalization of correctness (rather than trying to work purely axiomatically), see no reason why a program logic is more likely to be unsound than any interesting type system or static analysis in the literature (though opinions differ on just how likely that is. . .). As regards the second point, shallow embeddings are very convenient, especially for mechanization, but they do have the potential to miss the semantic subtleties of non-trivial languages. Whether or not one recognizes it, the embedding constitutes a denotational semantics that, if one is not extremely careful and the language is much more complex than while-programs, will be far from fully-abstract. Delegating the entailment used in the rule of consequence to implication in the metalogic therefore runs the risk of being incomplete for reasoning about behavioural properties of programs in the original language. It is unclear (at least to me) what the semantic import of, say, a relative completeness result factored through such a non-fully-abstract semantics is supposed to be.

There are many variations and improvements one might make to the logic, such as adding subtyping and polymorphism at the type level and adding other connectives to the program logic level. But it must be admitted that this system represents a rather odd point in the design space, as we have tried to keep the 'spirit' of traditional Hoare logic: pre and post conditions, first-order procedures and the use of classical predicate calculus to form assertions on a flat state. A generalisation to higher-order, with first-class code pointers, would bring some complexity, but also seems to offer some simplifications, such as the fact that one only needs preconditions as everything is in CPS. Another extension would be to more general dynamic allocation. Both first-class code pointers and heaps have been the objects of closely related work on semantics and types for both low-level and high-level languages (e.g. [9] and the references therein); transferring those ideas to general assertions on low-level code looks eminently doable. We would also like to generalize predicates to binary relations on states. Our ultimate goal is a relational logic for a low-level language into which one can translate a variety of high-level typed languages whilst preserving equational reasoning. We regard this system as a step towards that goal, rather than an endpoint in its own right.

We have not yet fully explored the ramifications of our semantic interpretation. One of its effects is to close the interpretation of extended label types with respect to an observational equivalence, which is a pleasant feature, but our inference system then seems unlikely to be complete. The links with work of Honda et al. [15] on observationally complete logics for state and higher-order functions deserve investigation. Note that the extent to which our extensional semantics entails a more naive intensional one depends on what test contexts one can write, and that these test contexts are not merely allowed to be untypable, but interesting ones all *are* untypeable: they 'go wrong' when a predicate fails to hold. There is an adjoint 'perping' operation that maps sets of configurations to subsets of $Stores \times Stacks$ and more inference rules (for example, involving conjunction) seem to be valid for state assertions that are closed, in the sense that $[\![E]\!] = [\![E]\!]^{\top\top}$. We could impose this closure by definition, or by moving away from classical logic for defining the basic assertions over states.

References

[1] M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In *Proc. 7th International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, 1997.

[2] A. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[3] A. Appel and A. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL)*, 2000.

[4] A. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5), 2001.

[5] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In *Proc. 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3223 of *Lecture Notes in Computer Science*, 2004.

[6] F. Bannwart and P. Muller. A program logic for bytecode. In *Proc. 1st Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, April 2005.

[7] N. Benton. A typed logic for stacks and jumps. Draft Note, March 2004.

[8] N. Benton. A typed, compositional logic for a stack-based abstract machine. Technical Report MSR-TR-2005-84, Microsoft Research, June 2005.

[9] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *Proc. 7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 3461 of *Lecture Notes in Computer Science*, 2005.

[10] J. Borgström. Translation of smart card applications for formal verification. Masters Thesis, SICS, Sweden, 2002.

[11] L. Cardelli. Program fragments, linking, and modularization. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[12] S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(6), 1999.

[13] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *Proc. 28th ACM Symposium on Principles of Programming Languages (POPL)*, 2001.

[14] N. A. Hamid and Z. Shao. Interfacing Hoare logic and type systems for foundational proof-carrying code. In *Proc. 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3223 of *Lecture Notes in Computer Science*, 2004.

[15] K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *Proc. 20th IEEE Symposium on Logic in Computer Science (LICS)*, 2005.

[16] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In *3rd International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2000.

[17] L. Jia, F. Spalding, D. Walker, and N. Glew. Certifying compilation for a language with stack allocation. In *Proc. 20th IEEE Symposium on Logic in Computer Science (LICS)*, 2005.

[18] T. Kleymann. Hoare logic and auxiliary variables. Technical Report ECS-LFCS-98-399, LFCS, University of Edinburgh, 1998.

[19] S. Lindley and I. Stark. Reducibility and $\top\top$ lifting for computation types. In *Proc. 7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 3461 of *Lecture Notes in Computer Science*, 2005.

[20] G. Morrisett, A. Amal, and M. Fluet. L3: A linear language with locations. In *Proc. 7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 3461 of *Lecture Notes in Computer Science*, 2005.

[21] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1), 2002.

[22] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3), 1999.

[23] G. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[24] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[25] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proc. 10th Annual Conference of the European Association for Computer Science Logic (CSL)*, volume 2142 of *Lecture Notes in Computer Science*, 2001.

[26] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In *Proc. Formal Methods Europe (FME)*, volume 2391 of *Lecture Notes in Computer Science*, 2002.

[27] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*. CUP, 1998.

[28] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In *Proc. 8th European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, 1999.

[29] C. Quigley. A programming logic for Java bytecode programs. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2758 of *Lecture Notes in Computer Science*, 2003.

[30] C. L. Quigley. *A Programming Logic for Java Bytecode Programs*. PhD thesis, University of Glasgow, Department of Computing Science, 2004.

[31] J. C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, 1982.

[32] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proc. 9th European Symposium on Programming (ESOP)*, volume 1782 of *Lecture Notes in Computer Science*, 2000.

[33] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symposium on Principles of Programming Languages (POPL)*, 1998.

[34] D. von Oheimb. Hoare logic for mutual recursion and local variables. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *Lecture Notes in Computer Science*, 1999.

[35] J. Vouillon and P.-A. Mellies. Semantic types: A fresh look at the ideal model for types. In *Proc. 31st ACM Symposium on Principles of Programming Languages (POPL)*, 2004.

[36] D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .NET common language runtime. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL)*, 2004.

A New Occurrence Counting Analysis for BioAmbients

Roberta Gori[1] and Francesca Levi[2]

[1] Department of Computer Science, University of Pisa, Italy
gori@di.unipi.it
[2] DISI, University of Genova, Italy
levifran@disi.unige.it

Abstract. This paper concerns the application of formal methods to biological systems, modelled specifically in BioAmbients [30]. BioAmbients [30] is a variant of the Mobile Ambients (MA)[7] calculus, designed precisely for more faithfully capturing basic biological concepts. We propose a new static analysis for BioAmbients which computes approximate information about the run-time behaviour of a system. The analysis is derived following the abstract interpretation approach and introduces two main novelties with respect to the analyses in literature [25,24,26,27]: (i) it records information about the number of occurrences of objects; (ii) it maintains more detailed information about the possible contents of ambients, at any time. In this way, the analysis gives substantially more precise results and captures both the quantitative and causal aspect which are really important for reasoning on the temporal and spatial structure of biological systems. The interest of the analysis is demonstrated by considering a few simple examples which point out the limitations of the existing analyses for BioAmbients.

Keywords: *Mobile Ambients and BioAmbients calculus, static analysis, abstract interpretation.*

1 Introduction

In the past few years several models, originally developed by computer scientists for describing systems of interacting components, have been successfully used for describing biological systems. This is an exciting and interesting application especially because the simulation and verification tools, designed for these formal models, can be used for understanding the behavior of complex biological systems. Such verification techniques can offer biologists very important instruments to replace expensive experiments in vitro or guide the biologists in their experiments by making predictions on the possible results.

Several models and languages, adequate for modeling different aspects of biological systems, have been proposed. They include traditional specification languages for concurrent and reactive systems [21,16,15] and also process calculi, designed for modeling distributed and mobile systems, which can successfully describe both the molecular and biochemical aspect. This approach is pioneered by the application of stochastic π-calculus [31,29], see for example the modeling of the RTK/MAPK pathway. New process calculi have also been proposed in order to faithfully model biological structures such as compartments and membranes, which play a key role in the organization of biomolecular systems. Among them, BioAmbients [30], Beta-Binders [28], and Brane calculi [6].

BioAmbients (BA) is a variant of Mobile Ambients (MA)[7], a very popular calculus designed to model distributed and mobile processes. The key concept of MA is that of *ambient*. An ambient represents a bounded location where computation happens; ambients are organized into a hierarchy, that can be dynamically modified as a consequence of an ambient movement or dissolution. The concepts of ambient and of ambient movement permit to naturally represent important aspects of molecular systems, such as localization, compartmentalization and hierarchy. With the aim of better capturing basic biological concepts, minor modifications are introduced in BA with respect to MA. Ambients are nameless; the primitive for opening is replaced by a primitive of merge, which realizes the fusion of two ambients; capabilities have corresponding co-capabilities; new primitives for communication and choice are introduced.

For BA both verification and simulation methods have been proposed that are essential for a practical application of the model. A stochastic simulation tool has been implemented by extending that of biochemical stochastic π-calculus [31,29]; tools for automatic verification include model checking [19,20] and static analysis [25,24,26,27]. *Static analysis* is a formal technique for computing safe approximations of the system (run-time) semantics, and it has been typically applied in the MA setting for verifying security properties. In our opinion, this technique of approximation is essential for dealing with the intrinsic complexity of biological systems.

The proposed analyses are obtained by naturally adapting to its variant BA existing Control Flow Analyses in Flow Logic style of MA. More in details, the analysis of [25] is derived from [23] and predicts the run-time behavior of processes, by giving information about the evolution of the ambients hierarchy, and about which capabilities may be exercised inside any ambient. The proposals of [24,27] refine the analysis of [23] by introducing more information about the possible shape of processes and about the context, along the lines of various analyses for π-calculus or MA [5,12,17].

These analyses give an over-approximation of the behavior of a process, and as usual guarantee invariant properties showing that certain events *will not happen* in *each* state of the system. In particular, they can be applied to establish whether an ambient *will never* end up inside another one; and similarly whether a capability *will never* be exercised inside a given ambient. This kind of information is crucial, when considering security guarantees, and also in the setting of biological systems; for example, in [27] this is enough for distinguishing a system describing a normal LDL degradation process from one presenting mutations or defects.Nonetheless, we believe that different and more detailed kinds of information would be very useful for biologists in order to argue about the spatial and temporal evolution of biological systems.

First, we observe that *quantitative information* plays an essential role in modeling and observing biological systems, as demonstrated by the following example.

Example 1. The system described below (inspired from the porin example in [30]) models the movement of molecules across membrane-bound compartments, specifically a cell. The cell and the molecules are described by ambients, labeled *cell* and *mol*, respectively; their local processes, e.g. P and M, describe their possible interactions.

$$\begin{aligned} SYS &::= [M]^{mol} \mid \ldots \mid [M]^{mol} \mid [P]^{cell} \\ M &::= \texttt{rec}X.\, \texttt{in}\, m.\, \texttt{out}\, n.\, X \\ P &::= \texttt{rec}Y.\, (\overline{\texttt{in}}\, m.\, Y + \overline{\texttt{out}}\, n.\, Y) && (1) \\ P &::= \texttt{rec}Y.\, \overline{\texttt{in}}\, m.\, \overline{\texttt{out}}\, n.\, Y && (2) \end{aligned}$$

Process M models the ability of molecules of entering and exiting from the membrane, any number of times. The complementary process P gives the permission to ambients *mol* to enter and exit from the *cell*, and thus regulates the crossing of the membrane. Specifically, when process (1) is running inside *cell*, then, at any time, a molecule can exit from or enter inside *cell*. Therefore, when several molecules are present, as described in process SYS, the cell may contain *any number* of molecules. By contrast, when process (2) is running inside *cell*, then, no other molecule can enter, after one has entered inside the *cell*. Thus, no matter how many molecules are present, *just one* molecule can reside inside the cell, at the time.

It is clear that processes (1) and (2) produce a substantially different behaviour for ambient *cell*. Unfortunately, the existing analyses for BA [25,24,26,27] can not capture this relevant difference; in fact, in both cases, they report that ambients *mol may reside inside ambients cell* without giving any information about the possible number of occurrences. Even occurrence counting analyses of MA [17,14] would be too coarse to model this difference, because they are designed for approximating the number of ambients which occur in the *whole* system. □

Another limitation of the existing analyses [25,24,26,27] is that they do not maintain sufficiently precise information about the possible contents of ambients, at any computation step. This has serious consequences on their ability to capture *causality aspects* which are essential for understanding the temporal and spatial structure of biological systems, such as pathways and networks of proteins. This point is illustrated by the following example.

Example 2. The system describes a simplified version of a bi-substrate enzymatic reaction, modeled as the movement of molecular ambients [30], where two molecules mol_1 and mol_1 interact with an *enzyme*.

$SYS' = [M_1]^{mol_1} \mid \ldots \mid [M_1]^{mol_1} \mid [M_2]^{mol_2} \mid \ldots \mid [M_2]^{mol_2} \mid [E]^e \mid \ldots \mid [E]^e$
$M_1 ::= \mathtt{rec}Y.\, \mathtt{in}\, m_1.\, (\mathtt{out}\, n_1.\, P_1 + \mathtt{out}\, q_1.\, Y)$
$M_2 ::= \mathtt{rec}Y.\, \mathtt{in}\, m_2.\, (\mathtt{out}\, n_2.\, P_2 + \mathtt{out}\, q_2.\, Y)$
$E ::= \mathtt{rec}Y.\, \overline{\mathtt{in}}\, m_1.\, \overline{\mathtt{in}}\, m_2.\, (\overline{\mathtt{out}}\, n_2.\, \overline{\mathtt{out}}\, n_1.\, Y + \overline{\mathtt{out}}\, q_2.\, \overline{\mathtt{out}}\, q_1.\, Y)$

The enzyme and its substrates are modeled by ambients, labeled e, mol_1 and mol_2, respectively. Processes M_1 and M_2 describe the movements of ambients mol_1 and mol_2, respectively; process E describes how the molecules bind to the enzyme and how their products are released. Specifically, the enzyme-substrate binding is modeled as entry of the substrate ambient inside the enzyme ambient, and it follows a precise order (mol_1 and then mol_2). When both molecules are inside the enzyme there are two possible evolutions: both molecules either exit unbind or exit and release their products P_1 and P_2 (these steps follow the inverse order). This process can iterate forever.

This enzymatic reaction has a crucial feature. Not only the binding of both substrates is necessary for the release of their products, but also it has to follow a precise order. This can be formalized by the following property: for each state the binding of mol_1 with e (shown by the presence of mol_1 inside e) is *necessary* for the binding of mol_2 with e (shown by the presence of mol_2 inside e). Such a property cannot be proved with the existing analyses for BA [25,24,26,27], which give too coarse information about the possible run-time nesting of ambients. In fact, they report that both ambients mol_1 and mol_2 may reside inside ambient e without saying whether the presence of one molecule depends on that of the other.

Moreover, a typical way to test whether both substrates mol_1 and mol_2 are *necessary* for the release of the products is to simulate an experiment where either mol_1 or mol_2 are removed. Unfortunately, using the analyses of [25,24,26,27] it is not possible to observe a change in the release of the products. □

Based on these motivations we propose a new analysis for BA following the *Abstract Interpretation* [9,10] approach to program analysis, more specifically in the style of previous proposals for MA [14,13,17]. The analysis refines the existing analyses of BA [25,24,26,27] by introducing two (strictly related) novelties: (i) quantitative information is modeled by recording information about the number of occurrences of ambients and processes which may appear in any location; (ii) more detailed information about the possible contents of ambients, at any time, is obtained by pushing forward the idea of continuations proposed in [17]. In this way, we obtain a more informative analysis which can be successfully applied to prove the properties of Examples 1 and 2.

This gain in precision is obviously paid in terms of complexity (in the worst case, the analysis is exponential); by contrast, the existing analyses [25,24,26,27] are associated with polynomial time algorithms. A great advantage of the abstract interpretation theory is that it offers the possibility to systematically define further approximations (e.g. new weaker analyses) by means of *widening operators* [11]. We show that this approach can be profitably applied also to our analysis by introducing a simple parametric widening which turns out to be polynomial in the size of a chosen partition of abstract labels. We then apply the widening to Example 2 for showing that it still gives better results w.r.t. the existing analyses [25,24,26,27].

The paper is organized as follows. Section 2 introduces the syntax and the semantics of the BioAmbients calculus. In Section 3 we presents our analysis and in Section 4 the corresponding widening operator.

2 Syntax and Semantics

For lack of space, we consider here a simplified version of BioAmbients [30] without communication primitives; the analysis can be extended in a simple way to the full calculus.

In Control Flow Analysis (see for instance [23,12]) typically processes are labeled and α-conversion is treated in a particular way, based on a given partition of labels and names. This modification supports simpler specifications of abstractions. We therefore consider the following sets of names and labels. Let $\mathcal{N}$ (ranged over by $n, m, h, k, \ldots$) be the set of *names* such that $\mathcal{N} = \uplus_{i=1}^{\omega} \mathcal{N}_i$, where $\uplus$ denotes disjoint union and each $\mathcal{N}_i$ is an infinite set. Similarly, let $\mathcal{L}$ (ranged over by $\lambda, \mu, \ldots$) be the set of *labels* such that $\mathcal{L} = \uplus_{i=1}^{\omega} \mathcal{L}_i \cup \{\top\}$, where each $\mathcal{L}_i$ is an infinite set and $\top$ is a distinct symbol used to model the outermost ambient. Moreover, we consider *composite labels* (in the following referred to as labels) $\widehat{\mathcal{L}} = \wp(\mathcal{L}) \setminus \emptyset$. We adopt meta-variables $\Psi, \Gamma, \Delta, \ldots$ to range over $\widehat{\mathcal{L}}$ and we use for simplicity λ for the singleton $\{\lambda\}$. We also consider a set of recursion variables $\mathcal{V}$ (ranged over by $X, Y, Z, \ldots$).

The syntax of (labelled) *processes* is defined in Table 1. The constructs for inactivity, parallel composition, restriction are standard (see for instance π-calculus [22]). The inactive process is denoted by 0; parallel composition is denoted by $P \mid Q$; the restriction operator, denoted by $(\nu n)\, P$, creates a new name n with scope P. Operator $\mathtt{rec}X^{\lambda}.\, P$ defines a recursive process (for convenience we adopt recursion in place of standard replication $!P$). Specific to the ambient calculi, are the ambient construct, $[P]^{\Psi}$, and the capability prefix $M^{\lambda}.\, P$, where M

Table 1. BioAmbients Processes

M,N::=		(*capabilities*)
	$\mathtt{in}\, n$	enter
	$\overline{\mathtt{in}}\, n$	co-enter
	$\mathtt{out}\, n$	exit
	$\overline{\mathtt{out}}\, n$	co-exit
	$\mathtt{merge}\, n$	merge
	$\overline{\mathtt{merge}}\, n$	co-merge

P,Q::=		(*processes*)
	0	inactivity
	$(\nu n)\, P$	restriction
	$P \mid Q$	parallel composition
	X	recursion variable
	$\mathtt{rec}X^{\lambda}.\, P$	recursive process
	$[P]^{\Psi}$	ambient
	$M^{\lambda}.\, P$	capability prefix
	$\Sigma_{i \in I}^{\lambda} M_i.\, P_i$	capability choice

is an action or co-action[1]. Specifically, process $[P]^{\Psi}$ defines an ambient (labelled) Ψ where process P runs. Finally, process $\Sigma^{\lambda}_{i\in I} M_i.\, P_i$ defines a capability choice primitive with the obvious meaning.

For processes we adopt standard syntactical conventions. We often omit the trailing 0 in processes, and we assume that parallel composition has the least syntactic precedence. The operator $(\nu n)P$ acts as static binder for name n, and thus produces the standard notion of free and bound names of a process; similarly, $\texttt{rec}X.\, P$ is a binder for X with scope P. A process is *closed on recursion variables* if it has no free recursion variables. In the following, we assume that processes are closed on recursion variables. Moreover, since processes are labelled, with $\Lambda(P)$ we indicate the set of labels of a process P.

Table 2. Reduction Rules of BioAmbients

$[+\texttt{in}\, m^{\lambda}.\, P \mid Q]^{\Psi} \mid [+\overline{\texttt{in}}\, m^{\mu}.\, R \mid S]^{\Delta} \to [[P \mid Q]^{\Psi} \mid R \mid S]^{\Delta}$	(In)
$[[+\texttt{out}\, m^{\lambda}.\, P \mid Q]^{\Psi} \mid +\overline{\texttt{out}}\, m^{\mu} R \mid S]^{\Delta} \to [P \mid Q]^{\Psi} \mid [R \mid S]^{\Delta}$	(Out)
$[+\texttt{merge}\, m^{\lambda}.\, P \mid Q]^{\Psi} \mid [+\overline{\texttt{merge}}\, m^{\mu}.\, R \mid S]^{\Delta} \to [P \mid Q \mid R \mid S]^{\Psi\cup\Delta}$	(Merge)
$\texttt{rec}X^{\lambda}.\, P \to P[\texttt{rec}X^{\lambda}.\, P/X]$	(Rec)
$P \to Q \Rightarrow (\nu n)\, P \to (\nu n)\, Q$	(Res)
$P \to Q \Rightarrow P \mid R \to Q \mid R$	(Par)
$P \to Q \Rightarrow [P]^{\Psi} \to [Q]^{\Psi}$	(Amb)
$(P' \to Q',\ P \equiv P',\ Q' \equiv Q) \Rightarrow P \to Q$	(Cong)

As usual, we identify processes which are α-convertible, that is that can be made syntactically equals by a change of bound names. In typical Control Flow Analysis style [12,17], we however discipline α-conversion by assuming that a bound name m can be replaced only with a name n provided that $n, m \in \mathcal{N}_i$. Similarly, we also identify *re-labeled* processes, i.e. processes that can be made syntactically equals by changing labels, requiring that a label λ can be replaced with a label μ, provided that $\lambda, \mu \in \mathcal{L}_i$.

The semantics of BA is given in the form of a standard reduction relation; the rules are reported in Table 2. In order to compact several rules together we introduce a special notation for capability prefix and capability choice. We write $+M^{\lambda}.\, P$ to denote both process $M^{\lambda}.\, P$ and process $\Sigma^{\lambda}_{i\in I} M_i.\, Q_i$, where $M = M_i$ and $P = Q_i$ for some $i \in I$.

[1] Notice that we adopt a notation for coactions in the style of Safe Ambients [18] in place of the standard one.

The reduction axioms (In), (Out) and (Merge) define the basic interactions; they model the movement of an ambient, in or out, of another ambient and the merge of two ambients. They differ from those of MA mainly because ambients are nameless (labels are attached to processes as comments and do not influence the interaction). Moreover, the primitive *merge* replaces the standard primitive of opening. Notice that, when two ambients labelled Ψ and Δ are merged, the new ambient is labelled $\Psi \cup \Delta$ showing that it is the result of their fusion. Another difference with MA, common instead with its variant Safe Ambients [18], is that we prefer to view the unfolding of recursion as a reduction rule, e.g. (Rec), rather than as a step of structural congruence.

The inference rules (Res), (Par), (Amb) and (Cong) are standard; they handle reductions in contexts and permit to apply structural congruence. Structural congruence is needed to bring the participants of a potential interaction into contiguous positions; it includes standard rules for commuting the positions of components appearing in parallel and in a choice, and rules for stretching the scope of a restrictions. For lack of space, we omit the presentation of structural congruence (e.g. relation $\equiv$) and we refer to [30]. In the following, we say that a process P is *active* if either $P = \Sigma^{\lambda}_{i \in I} M_i.\, Q_i$, $P = M^{\lambda}.\, Q$ or $P = \mathtt{rec}X^{\lambda}.\, P$. Moreover, we use $\mathcal{P}$ and $\mathcal{AP}$ to denote the set of processes and the subset of active processes, respectively.

The collecting semantics. The collecting semantics is defined as the least fixed-point of a function, which collects all the states (namely processes) reachable from the initial process. The *concrete domain* is therefore $\mathcal{A} = (\wp(\mathcal{P}), \subseteq)$.

Definition 1 (Collecting Semantics). *Let $P \in \mathcal{P}$ be a process such that $\top \notin \Lambda(P)$. We define $\mathfrak{S}_{Coll}[\![P]\!] =$ lfp $F(P)$ for the function $F : \mathcal{P} \to (\mathcal{A} \to \mathcal{A})$ such that $F(P) = \Psi_P$ and, for $Ss \in \wp(\mathcal{P})$,*

$$\Psi_P(Ss) = \{P\} \bigcup_{\{P_2 | P_1 \to P_2,\ P_1 \in Ss\}} \{P_2\}.$$

3 The Abstraction

Our analysis is designed to prove properties that are true in all the states reachable from the initial state. To this aim, it computes an over-approximation of the following information about any reachable state: for each ambient, which ambients may be contained and which capabilities may be exercised inside, and their number of occurrences. Following the abstract interpretation approach of [17] we define the analysis by giving the abstract states (the abstract processes) and the abstract transitions (the abstract reduction steps among processes). To formally prove the correctness of the analysis, we introduce a corresponding abstract domain, equipped with an ordering expressing precision of approximations, and we formalize its relation with the concrete one through a Galois connection [9,10].

The abstraction is parametric with respect to the choice of *abstract names* and *labels*, defined by a partition of names and labels. For these purposes, we first consider an abstract partition of labels $\mathcal{L}$, given by $\mathcal{L}^\circ = \uplus_i \mathcal{L}_i^\circ \cup \{\top\}$, where $i \in \{1, \ldots, h\}$ for some h, $\mathcal{L}_i^\circ$ is a (possible) infinite set of labels, $\uplus_i \mathcal{L}_i^\circ \cup \{\top\} = \mathcal{L}$ and $\mathcal{L}^\circ$ is congruent with $\mathcal{L}$, i.e., $\lambda, \mu \in \mathcal{L}_i$ implies that $\lambda, \mu \in \mathcal{L}_j^\circ$ for some j. We consider, then, *abstract labels* $\widehat{\mathcal{L}}^\circ = \wp(\mathcal{L}^\circ_{/\cong}) \setminus \emptyset$ (ranged over by $\Psi^\circ, \Gamma^\circ, \Delta^\circ, \ldots$), where $\cong$ is the obvious equivalence induced by the partition. For names we proceed in a similar way by considering an abstract partition of names $\mathcal{N}^\circ = \uplus_i \mathcal{N}_i^\circ$, where $i \in \{1, \ldots, h\}$ for some h, $\uplus_i \mathcal{N}_i^\circ = \mathcal{N}$, such that $\mathcal{N}^\circ$ is congruent with $\mathcal{N}$, i.e., $n, m \in \mathcal{N}_i$ implies that $n, m \in \mathcal{N}_i^\circ$. We therefore consider *abstract names* $\widehat{\mathcal{N}}^\circ = \wp(\mathcal{N}^\circ_{/\cong} *) \setminus \emptyset$ (ranged over by $A^\circ, B^\circ, C^\circ, \ldots$). For convenience, we assume that Δ° stands for the abstract label (namely its equivalence class) corresponding to label Δ; similarly for abstract names.

The abstract partitions of names and labels naturally induce a corresponding notion of abstract processes; built following the syntax of Table 1 by using names $\widehat{\mathcal{N}}^\circ$ and labels $\widehat{\mathcal{L}}^\circ$. As usual, $\mathcal{P}^\circ$ and $\mathcal{AP}^\circ$ stands for the set of abstract and active abstract processes, respectively. Similarly, P° stands for the abstract process corresponding to P (this is obtained in the obvious way).

Abstract domain and Galois connection. Abstract states are the key concept behind the abstraction and are designed precisely to represent approximate information about "concrete" states (e.g. processes) according to the following intuitive ideas. An *abstract state* reports: (i) the abstract labels of the ambients that may appear; and (ii) for each of them, one or more *configurations* describing the possible contents of the ambients with that label. More in details, a configuration contains the *abstract labels* of the ambients and the *active abstract processes*, which may appear at top- level, and their number of occurrences. For representing occurrence counting information, we adopt the following set $\mathcal{M}$ $=\{0, 1, [0-\omega], [1-\omega]\}$. Each $\mathtt{m} \in \mathcal{M}$ denotes a *multiplicity*, with the following meaning: 0 and 1 indicate zero and exactly one respectively, the interval $[1-\omega]$ at least one while the interval $[0-\omega]$ indicate 0 or more.

Example 3. Consider the system (already described in Example 1),

$$\begin{aligned} SYS &::= [M]^{mol} \mid \ldots \mid [M]^{mol} \mid [P]^{cell} \\ M &::= \quad \mathtt{rec}X.\, \mathtt{in}\, m.\, \mathtt{out}\, n.\, X \\ P &::= \quad \mathtt{rec}Y.\, \overline{\mathtt{in}}\, m.\, \overline{\mathtt{out}}\, n.\, Y \end{aligned}$$

Moreover, assume that abstract names and labels are defined by the following equivalence classes $\{\{n, m\}\}$ (ranged over by m) and $\{\{mol\}, \{cell\}\}$, respectively. With respect to this partition of labels and names, the *best approximation* of SYS is given by the following abstract state (graphically represented also in Figure 1)

$$\begin{array}{l} S^\circ = \{(\top, C_0^\circ), (mol, C_1^\circ), (cell, C_2^\circ)\} \quad C_0^\circ = \{(mol, [1-\omega]), (cell, 1)\} \quad C_1^\circ = \{(M^\circ, 1)\} \\ C_2^\circ = \{(P^\circ, 1)\} \quad M^\circ ::= \mathtt{rec}X.\, \mathtt{in}\, m.\, \mathtt{out}\, m.\, X \qquad P^\circ ::= \mathtt{rec}Y.\, \overline{\mathtt{in}}\, m.\, \overline{\mathtt{out}}\, m.\, Y \end{array}$$

Configuration C_0° reports information about the possible internal process of ambient $\top$ (a special symbol representing the outermost ambient). More in

$$\begin{tikzcd}
& \top \arrow[dl, dotted] \arrow[dr, dotted] & \\
(\mathit{mol}, [1-\omega]) \arrow[d] & & (\mathit{cell}, 1) \arrow[d] \\
(M^\circ, 1) & & (P^\circ, 1)
\end{tikzcd}$$

Fig. 1. State S° graphically

details, pair $(\mathit{cell}, 1)$ says that *exactly one* ambient *cell* may appear at top-level, while pair $(\mathit{mol}, [1-\omega])$ says that *at least one* ambient *mol* may appear at top-level. Ambients *cell* and *mol* may appear in parallel inside ambient $\top$; this is shown by a dotted line that connects these ambients with their father in Figure 1. Configurations C_1° and C_2° describe the possible internal processes of ambients *mol* and *cell*, respectively. In C_1° pair $(M^\circ, 1)$ says that, inside *any* ambient *mol*, *exactly* one process abstracted by M° may be running. In this sense, the counting of occurrences is local, being $[1-\omega]$ the global number of occurrences of processes M°. Similarly, in C_2° pair $(P^\circ, 1)$ says that, inside *any* (in this case one) ambient *cell*, *exactly* one process abstracted by P° may be running.

Consider then a minor modification of SYS, where more than one ambient *cell* may appear, $SYS_1 = [M]^{mol} \mid \ldots \mid [M]^{mol} \mid [P]^{cell} \mid \ldots \mid [P]^{cell}$. Now the best approximation is

$$S_1^\circ = \{(\top, \{(\mathit{mol}, [1-\omega]), (\mathit{cell}, [1-\omega])\}), (\mathit{mol}, C_1^\circ), (\mathit{cell}, C_2^\circ)\}.$$

The only difference between S° and S_1° concerns the multiplicity of ambients *cell*, which is now $[1-\omega]$. It is clear that state S_1° is also a correct approximation for process SYS; it is however less precise than S°, which predicts *exactly* one occurrence of ambients *cell* at top-level.

It is worth noticing that in abstract states S° and S_1° exactly one configuration describes the possible internal processes of each abstract label (and thus of the related ambients). It may be convenient however to adopt several different configurations, as illustrated by the following system,

$$SYS_2 ::= [M]^{mol} \mid \ldots \mid [M]^{mol} \mid [\overline{\mathtt{out}}\, m.\, P \mid [\mathtt{out}\, m.\, M]^{mol}]^{cell}$$

This process is a derivative of SYS and describes the situation where: one ambient *mol* has moved inside ambient *cell* and is ready to exit; the remaining ambients *mol* are still in the initial situation. Process SYS_2 could be approximated by the following abstract state,

$$\begin{array}{l}
S_2^\circ = \{(\top, C_0^\circ), (\mathit{mol}, C_4^\circ), (\mathit{cell}, C_5^\circ)\} \\
C_5^\circ = \{(\overline{\mathtt{out}}\, m.\, P^\circ, 1), (\mathit{mol}, 1)\} \quad C_4^\circ = \{(M^\circ, [0-\omega]), (\mathtt{out}\, m.\, M^\circ, [0-\omega])\}
\end{array}$$

Configuration C_5° describes the possible contents of ambients *cell* and shows that: exactly one ambient *mol* and exactly one process abstracted by $\overline{\mathtt{out}}\, m.\, P^\circ$ may appear; these processes may be running in parallel inside an ambient *cell*. Configuration C_4° reports the information about the possible contents of ambients

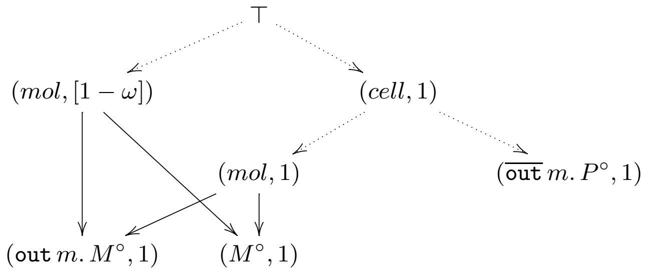

$\top$
$(mol, [1-\omega])$
$(cell, 1)$
$(mol, 1)$
$(\overline{\texttt{out}}\, m.\, P^\circ, 1)$
$(\texttt{out}\, m.\, M^\circ, 1)$
$(M^\circ, 1)$

Fig. 2. State S_3° graphically

mol; it describes both those at top-level (that contain the recursive process) and the one, residing inside ambient *cell* (where process $\texttt{out}\, m.\, M^\circ$ is running). The configuration says that, inside any ambient *mol*, zero or more processes abstracted by M° and $\texttt{out}\, m.\, M^\circ$ may appear (in particular they may be running in parallel). Notice that, since all the ambients *mol* are identified, the multiplicity, for each process, is $[0-\omega]$ showing that it may be the case that the process does not appear.

The information about ambients *mol* in state S_2° is rather approximate. Better results can be obtained by adopting distinct configurations to describe the different instances of ambients *mol*, as in the following abstract state,

$$S_3^\circ = \{(\top, C_0^\circ), (cell, C_5^\circ), (mol, C_6^\circ), (mol, C_7^\circ)\} \quad C_6^\circ = \{(M^\circ, 1)\} \quad C_7^\circ = \{(\texttt{out}\, m.\, M^\circ, 1)\}$$

In this case, ambients *mol* are described by two configurations, C_6° and C_7° . Their interpretation is that any ambient *mol*, contains *either* exactly one process abstracted by M° or exactly one process abstracted by $\texttt{out}\, m.\, M^\circ$. In this way, more precise information about the multiplicity of processes M° and $\texttt{out}\, m.\, M^\circ$ is achieved; also, it is possible to argue that the two processes cannot run in parallel inside the same instance of ambient *mol*. This state is graphically represented in Figure 2 where these processes are connected with their enclosing ambient *mol* by a plain line precisely for showing that they cannot be in parallel. □

We introduce the formal definitions. In the following, we use $\widehat{\mathcal{PL}} = \widehat{\mathcal{L}}^\circ \cup \mathcal{AP}^\circ$ to denote the set of abstract labels and abstract active processes; also, we use e to denote a generic element of $\widehat{\mathcal{PL}}$. Moreover, we use $(e, \mathtt{m})$ to denote a generic element of $\mathcal{E} = (\widehat{\mathcal{L}}^\circ \times \mathcal{M}) \cup (\mathcal{AP}^\circ \times \mathcal{M})$.

Definition 2 (Abstract States). *An* abstract state S° *is a set of pairs* (Ψ°, C°) *where* $C^\circ \in \wp(\mathcal{E})$ *is a* configuration*, such that: (i) if* $(e, \mathtt{m}), (e, \mathtt{m}') \in C^\circ$*, then* $\mathtt{m} = \mathtt{m}'$*; and (ii) for each* $(e, \mathtt{m}) \in C^\circ$*,* $\mathtt{m} \neq 0$.

Notice that in configurations, no pair $(e, 0)$ can appear, recording explicitly that there are no occurrences of element e. However, in the following with an abuse of notation we may write $(e, 0) \in C^\circ$ in place of $(e, \mathtt{m}) \notin C^\circ$ for any $\mathtt{m} \in \mathcal{M}$.

Table 3. Occurrence Counting

$$\begin{tikzcd}
& {[0-\omega]} \arrow[dl, no head] \arrow[ddr, no head] & \\
{[1-\omega]} \arrow[d, no head] & & \\
1 \arrow[dr, no head] & & 0 \arrow[dl, no head] \\
& \bot &
\end{tikzcd}$$

$+^\circ$	0	1	$[1-\omega]$	$[0-\omega]$
0	0	1	$[1-\omega]$	$[0-\omega]$
1	1	$[1-\omega]$	$[1-\omega]$	$[1-\omega]$
$[1-\omega]$	$[1-\omega]$	$[1-\omega]$	$[1-\omega]$	$[1-\omega]$
$[0-\omega]$	$[0-\omega]$	$[1-\omega]$	$[1-\omega]$	$[0-\omega]$

$-^\circ$	1
0	0
1	0
$[1-\omega]$	$[0-\omega]$
$[0-\omega]$	$[0-\omega]$

This notation simplifies the definition of some operators over configurations and states. In the following, $\mathcal{S}^\circ$ and $\mathcal{C}^\circ$ stand for the set of abstract states and of configurations, respectively.

Following the intuitive ideas explained in Example 3 we introduce two information orders on configurations and abstract states which formalize precision of approximations. To this aim, we assume that the domain $\mathcal{M}$ of multiplicity comes equipped with the obvious (information) order $\leq_m$ and with the set of operations $+^\circ$ and $-^\circ$, reported in Table 3.

Definition 3 (Ordering).

- *We say that* $C_1^\circ \leq^c C_2^\circ$ *iff, for each* $(e, \mathtt{m}) \in C_1^\circ$ *there exists* $(e, \mathtt{m}') \in C_2^\circ$ *such that* $\mathtt{m} \leq_m \mathtt{m}'$*;*
- *We say that* $S_1^\circ \leq^s S_2^\circ$ *iff, for each* $(\Psi^\circ, C_1^\circ) \in S_1^\circ$*, there exists* $(\Psi^\circ, C_2^\circ) \in S_2^\circ$ *such that* $C_1^\circ \leq^c C_2^\circ$*.*

$\leq^s$ is a pre-order. We consider the order $\subseteq^\circ$ induced by the pre-order $\leq^s$, namely the order obtained considering classes of abstract states modulo the equivalence induced by $\leq^s$. For a sake of simplicity in the rest of the paper the domain $\mathcal{S}^\circ_{/\cong^s}$ and the equivalence class $[S^\circ]_{\cong^s}$ will be simply indicated by $\mathcal{S}^\circ$ and S° respectively.

Given the ordering over abstract states, it is immediate to define the *abstract domain*, $\mathcal{A}^\circ = (\mathcal{S}^\circ, \subseteq^\circ)$. Notice that the concrete domain records sets of states (e.g. processes); while in the abstract domain only one abstract states collects all the information.

The relation between the concrete and the abstract domain is formalized by establishing a Galois connection. To this aim, we first introduce a function that, given a process reports its *best approximation*, that is the best abstract state which has enough information about the process. This is derived along the lines of the intuitive ideas explained in Example 3. More in details, given a process P, we proceed as follows

Table 4. Abstract Translation Function

DRes$^\circ$	$\eta^\circ((\nu M^\circ)P^\circ)$	$= \eta^\circ(P^\circ)$	
DAmb$^\circ$	$\eta^\circ([P^\circ]^{\Delta^\circ})$	$= (\{(\Delta^\circ,1)\}, \delta^\circ(\Delta^\circ, P^\circ))$	
DZero$^\circ$	$\eta^\circ(0)$	$= (\emptyset,\ \emptyset)$	
DPar$^\circ$	$\eta^\circ(P_1 \mid P_2)$	$= ({C_1}^\circ \cup^+ {C_2}^\circ, {S_1}^\circ \cup^\circ {S_2}^\circ)$	$\eta^\circ(P_i) = (C_i^\circ, S_i^\circ)$ for $i \in \{1,2\}$
DRec$^\circ$	$\eta^\circ(\mathtt{rec}X^{\Psi^\circ}.\, P^\circ)$	$= (\{(\mathtt{rec}X^{\Psi^\circ}.\, P^\circ, 1)\}, \emptyset)$	
DPref$^\circ$	$\eta^\circ(M^{\Psi^\circ}.\, P^\circ)$	$= (\{(M^{\Psi^\circ}.\, P^\circ, 1)\}, \emptyset)$	
DSum$^\circ$	$\eta^\circ(\Sigma_{i\in I}^{\Psi^\circ} M_i^\circ.\, P_i^\circ)$	$= (\{(\Sigma_{i\in I}^{\Psi^\circ} M_i^\circ.\, P_i^\circ, 1)\}, \emptyset)$	

1. we take its abstract version P° where labels $\widehat{\mathcal{L}}$ and names $\widehat{\mathcal{N}}$ are replaced with their abstract versions $\widehat{\mathcal{L}}^\circ$ and $\widehat{\mathcal{N}}^\circ$, respectively;
2. we produce a representation of P° in terms of set of configurations where explicit information about the nesting of ambients and processes and about their quantities is properly introduced.

Formally, we define $\alpha^{aux} : \mathcal{P} \to \mathcal{S}^\circ$ as $\alpha^{aux}(P) = \delta^\circ(\top, P^\circ)$ where $\delta^\circ : (\widehat{\mathcal{L}}^\circ \times \mathcal{P}^\circ) \to \mathcal{S}^\circ$ is an *auxiliary translation function*, giving an abstract state representing the abstract process with respect to the label of the enclosing ambient (in this case $\top$).

The translation function $\delta^\circ : (\widehat{\mathcal{L}}^\circ \times \mathcal{P}^\circ) \to \mathcal{S}^\circ$ exploits an additional function $\eta^\circ : \mathcal{P}^\circ \to (\mathcal{C}^\circ \times \mathcal{S}^\circ)$, which intuitively gives: (i) an abstract configuration reporting the processes and ambients occurring at top level; (ii) an abstract state representing the internal ambients. Having in mind this interpretation we define

$$\delta^\circ(\Psi^\circ, P^\circ) = \{(\Psi^\circ, C^\circ)\} \cup S^\circ \quad \text{where} \quad \eta^\circ(P^\circ) = (C^\circ, S^\circ).$$

Function η° is reported in Table 4 and uses an operator $\cup^+$ between configurations, which simply realizes the union of two configurations by summing the multiplicity in the obvious way. Given $C_1^\circ, C_2^\circ \in \wp(\mathcal{E})$, we define

$$C_1^\circ \cup^+ C_2^\circ = \{(e, \mathtt{m}) \mid (e, \mathtt{m}_i) \in C_i^\circ,\ \text{for each } i \in \{1,2\},\ \mathtt{m} = \mathtt{m}_1 +^\circ \mathtt{m}_2\}.$$

As an example, it is not difficult to check that for the system of Example 3, we have $\delta^\circ(\top, SYS^\circ) = S^\circ$, where S° is the state of Figure 1.

Based on function α^{aux} it is immediate to derive the following abstraction and concretization functions between sets of processes and abstract states. This permits precisely to formalize when an abstract state is a *safe over-approximation* of a set of concrete states. We use $\cup^\circ$ between abstract states to denote the l.u.b. with respect to $\subseteq^\circ$; it realizes indeed the union of configurations.

Definition 4. *Let $Ss \in \wp(\mathcal{P})$ and $S^\circ \in \mathcal{S}^\circ$. We define $\alpha^\circ : \wp(\mathcal{P}) \to \mathcal{S}^\circ$ and $\gamma^\circ : \mathcal{S}^\circ \to \wp(\mathcal{P})$ as follows,*

$$\alpha^\circ(Ss) = \bigcup\nolimits^\circ_{P \in Ss} \alpha^{aux}(P) \quad \gamma^\circ(S^\circ) = \bigcup\nolimits_{\{P \mid \alpha^{aux}(P) \subseteq^\circ S^\circ\}} P$$

Theorem 1. *The pair of functions* $(\alpha^\circ, \gamma^\circ)$ *of Def. 4 is a Galois connection between* $\langle \mathcal{A}, \subseteq \rangle$ *and* $\langle \mathcal{A}^\circ, \subseteq^\circ \rangle$.

Abstract semantics. We introduce the *abstract transitions.* They use the following operators which realize the removal from a configuration of one occurrence of an object e and similarly of a set of objects. For $PL^\circ \subseteq \widehat{\mathcal{PL}}$ we have

$$C^\circ \backslash^\circ e = C^\circ \setminus \{(e, \mathtt{m})\} \cup \{(e, \mathtt{m} -^\circ 1)\} \qquad C^\circ \backslash^\circ PL^\circ = C^\circ \backslash^\circ_{e \in PL^\circ} e.$$

The *abstract transitions* are defined by the rules of Table 5; they realize the unfolding of recursion, the movements of ambients, in and out, and the merge of two ambients (reflecting rules (Rec), (In), (Out) and (Merge) of Table 2). Due to the implicit representation of parallel composition, ambient and restriction in abstract states there are no abstract transitions corresponding to the structural rules and to structural congruence of the reduction semantics.

Rule $\mathbf{Rec}^\circ$ models the unfolding of recursion and is applicable to a recursive process $T^\circ = \mathtt{rec}X^{\Delta^\circ}.P^\circ$ which is running inside an ambient labeled Γ° (this means that there exists a configuration C° for Γ° that contains process T°). The resulting abstract state is obtained by adding a configuration representing the ambient labeled Γ° where replication has been unfolded. More specifically, the configuration is $(C^\circ \backslash^\circ T^\circ) \cup^+ \delta^\circ(\Gamma^\circ, P^\circ[T^\circ/X])$ where the translation of the unfolded process is added and the recursive process T° is removed (according to their multiplicities).

The rules $\mathbf{In}^\circ$, $\mathbf{Out}^\circ$, $\mathbf{Merge}^\circ$ are similar; as an example we comment $\mathbf{In}^\circ$. The rule models the movement of an ambient labeled Ψ° inside an ambient labeled Δ°. It is applicable whenever: (i) they are possible siblings meaning that they may be enclosed, at the same time, inside an ambient (labeled Γ°); (ii) they offer the right action or coaction. Formally, it must be the case that: (i) there exists a configuration C° for Γ° which contains *both* Ψ° and Δ°; (ii) there exist configurations $C_1{}^\circ$ and $C_2{}^\circ$ for Ψ° and Δ°, where capabilities $\mathtt{in}\,M^\circ$ and $\overline{\mathtt{in}}\,M^\circ$ are ready to fire, respectively. If Ψ° and Δ° happen to be the same label, then the movement is possible only if their multiplicities are greater than 1.

The resulting abstract state is obtained as follows. Abstract configurations are added for modeling the ambients labeled Ψ° and Γ° which have participated to the movement:

1. $(C_1^\circ \backslash^\circ T^\circ) \cup^+ \delta^\circ(\Psi^\circ, P^\circ)$ describes the local process of ambient Ψ° and is obtained by removing the executed process T° and by adding its continuation (according to their multiplicities);
2. $(C_2^\circ \backslash^\circ T'^\circ) \cup^+ \delta^\circ(\Delta^\circ, Q^\circ) \cup^+ (\Psi^\circ, 1)$ describes the local process of ambient Δ° and is obtained similarly as in the previous case. The only relevant difference is that one occurrence of ambient Ψ° is added for modeling the movement.

Similarly, abstract configuration $C^\circ \backslash^\circ \Psi^\circ$ is added for the ambient labeled Γ°, taking into account precisely that one ambient labeled Ψ° has moved somewhere-else.

Table 5. Abstract Transitions

$$\textbf{Rec}^\circ \quad \frac{(\Gamma^\circ, C^\circ) \in S^\circ \qquad (T^\circ, \mathtt{m}) \in C^\circ \qquad T^\circ = \mathtt{rec}X^{\Delta^\circ}.\,P^\circ}{S^\circ \mapsto^\circ S^\circ \cup^\circ \{(\Gamma^\circ, (C^\circ \setminus^\circ T^\circ) \cup^+ \delta^\circ(\Gamma^\circ, P^\circ[T^\circ/X]))\}}$$

$$\textbf{In}^\circ \quad \frac{\begin{array}{l} (\Psi^\circ, C_1^\circ) \in S^\circ \qquad (T^\circ, \mathtt{m}_1) \in C_1^\circ \qquad T^\circ = +\mathtt{in}\, M^{\circ\,\Theta^\circ}.\,P^\circ \\ (\Delta^\circ, C_2^\circ) \in S^\circ \qquad (T'^\circ, \mathtt{m}_2) \in C_2^\circ \qquad T'^\circ = +\overline{\mathtt{in}}\, M^{\circ\,\Phi^\circ}.\,Q^\circ \\ (\Gamma^\circ, C^\circ) \in S^\circ \qquad (\Psi^\circ, \mathtt{m}_3) \in C^\circ \qquad (\Delta^\circ, \mathtt{m}_4) \in C^\circ \\ \Delta^\circ = \Psi^\circ \to \mathtt{m}_3 = \mathtt{m}_4 >_m 1 \end{array}}{S^\circ \mapsto^\circ \begin{array}{l} S^\circ \cup^\circ \{(\Psi^\circ, (C_1^\circ \setminus^\circ T^\circ) \cup^+ \delta^\circ(\Psi^\circ, P^\circ))\} \cup^\circ \\ \{(\Delta^\circ, (C_2^\circ \setminus^\circ T'^\circ) \cup^+ \delta^\circ(\Delta^\circ, Q^\circ) \cup^+ (\Psi^\circ, 1))\} \cup^\circ \{(\Gamma^\circ, C^\circ \setminus^\circ \Psi^\circ)\} \end{array}}$$

$$\textbf{Out}^\circ \quad \frac{\begin{array}{l} (\Psi^\circ, C_1^\circ) \in S^\circ \qquad (T^\circ, \mathtt{m}_1) \in C_1^\circ \qquad T^\circ = +\mathtt{out}\, M^{\circ\,\Theta^\circ}.\,P^\circ \\ (\Delta^\circ, C_2^\circ) \in S^\circ \qquad (T'^\circ, \mathtt{m}_2) \in C_2^\circ \qquad T'^\circ = +\overline{\mathtt{out}}\, M^{\circ\,\Phi^\circ}.\,Q^\circ \qquad (\Psi^\circ, \mathtt{m}_3) \in C_2{}^\circ \\ (\Gamma^\circ, C^\circ) \in S^\circ \qquad (\Delta^\circ, \mathtt{m}_4) \in C^\circ \end{array}}{S^\circ \mapsto^\circ \begin{array}{l} S^\circ \cup^\circ \quad \{(\Psi^\circ, (C_1^\circ \setminus^\circ T^\circ) \cup^+ \delta^\circ(\Psi^\circ, P^\circ))\} \cup^\circ \\ \{(\Delta^\circ, (C_2^\circ \setminus^\circ \{T'^\circ, \Psi^\circ\}) \cup^+ \delta^\circ(\Delta^\circ, Q^\circ))\} \cup^\circ \{(\Gamma^\circ, C^\circ \cup^+ (\Psi^\circ, 1))\} \end{array}}$$

$$\textbf{Merge}^\circ \quad \frac{\begin{array}{l} (\Psi^\circ, C_1^\circ) \in S^\circ \qquad (T^\circ, \mathtt{m}_1) \in C_1^\circ \qquad T^\circ = +\mathtt{merge}\, M^{\circ\,\Theta^\circ}.\,P^\circ \\ (\Delta^\circ, C_2^\circ) \in S^\circ \qquad (T'^\circ, \mathtt{m}_2) \in C_2^\circ \qquad T'^\circ = +\overline{\mathtt{merge}}\, M^{\circ\,\Phi^\circ}.\,Q^\circ \\ (\Gamma^\circ, C^\circ) \in S^\circ \qquad (\Psi^\circ, \mathtt{m}_3) \in C^\circ \qquad (\Delta^\circ, \mathtt{m}_4) \in C^\circ \\ \Delta^\circ = \Psi^\circ \to \mathtt{m}_3 = \mathtt{m}_4 >_m 1 \end{array}}{S^\circ \mapsto^\circ \begin{array}{l} S^\circ \cup^\circ \{(\Psi^\circ \cup \Delta^\circ, (C_2^\circ \setminus^\circ T'^\circ) \cup^+ (C_1^\circ \setminus^\circ T^\circ) \cup^+ \delta^\circ(\Psi^\circ, P^\circ) \cup^+ \delta^\circ(\Delta^\circ, Q^\circ))\} \cup^\circ \\ \{(\Gamma^\circ, (C^\circ \setminus^\circ \{\Delta^\circ, \Psi^\circ\}) \cup^+ (\Psi^\circ \cup \Delta^\circ, 1))\} \end{array}}$$

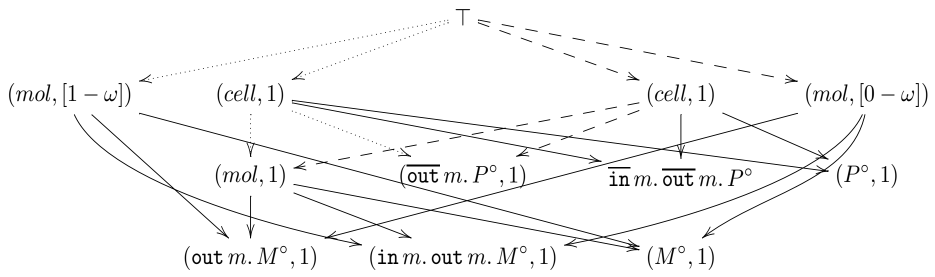

$\top$
$(mol, [1-\omega])$
$(cell, 1)$
$(cell, 1)$
$(mol, [0-\omega])$
$(mol, 1)$
$(\overline{\mathtt{out}}\, m.\, P^\circ, 1)$
$\overline{\mathtt{in}}\, m.\, \overline{\mathtt{out}}\, m.\, P^\circ$
$(P^\circ, 1)$
$(\mathtt{out}\, m.\, M^\circ, 1)$
$(\mathtt{in}\, m.\, \mathtt{out}\, m.\, M^\circ, 1)$
$(M^\circ, 1)$

Fig. 3. The analysis of SYS

The abstract semantics is then defined as the least fixed-point of a function, which computes starting from the abstraction of the initial process, an abstract state that is the union of all the reachable abstract states.

Definition 5 (The abstract semantics). *Let P be a process such that $\top \notin \Lambda(P)$. We define $\mathfrak{S}_{Coll^\circ}[\![P]\!] = \mathit{lfp}\ F^\circ(\alpha^{aux}(P))$ for the function $F^\circ : \mathcal{S}^\circ \to (\mathcal{A}^\circ \to \mathcal{A}^\circ)$ such that $F^\circ(S^\circ) = \Psi^\circ_{S^\circ}$ and*

$$\Psi^\circ_{S^\circ}(S_1^\circ) = S^\circ \cup^\circ \bigcup\nolimits^\circ_{\{S_2{}^\circ \mid S_1{}^\circ \mapsto^\circ S_2{}^\circ\}} S_2{}^\circ.$$

The following theorem shows the correctness of the analysis.

Theorem 2 (Safeness). *Let P be a process such that $\top \notin \Lambda(P)$. We have*

$$\alpha^{\circ}(\mathfrak{S}_{Coll}[\![P]\!]) \subseteq^{\circ} \mathfrak{S}_{Coll^{\circ}}[\![P]\!].$$

Example 4. We apply the abstraction to process SYS of Example 3 (and also Example 1). The analysis computes the following abstract state (also represented in Figure 3), namely $\mathfrak{S}_{Coll^{\circ}}[\![P]\!] = S^{\circ}_{SYS}$

$$S^{\circ}_{SYS} = \{(\top, C^{\circ}_0), (mol, C^{\circ}_1), (cell, C^{\circ}_2), (cell, C^{\circ}_3), (cell, C^{\circ}_4), (\top, C^{\circ}_5), (mol, C^{\circ}_6), (mol, C^{\circ}_7)\}$$

$$C^{\circ}_0 = \{(mol, [1-\omega]), (cell, 1)\} \qquad C^{\circ}_1 = \{(M^{\circ}, 1)\} \qquad C^{\circ}_2 = \{(P^{\circ}, 1)\}$$

$$C^{\circ}_3 = \{(\overline{\mathtt{in}}\, m.\, \overline{\mathtt{out}}\, m.\, P^{\circ}, 1)\} \quad C^{\circ}_4 = \{(\overline{\mathtt{out}}\, m.\, P^{\circ}, 1), (mol, 1)\}$$

$$C^{\circ}_5 = \{(mol, [0-\omega]), (cell, 1)\} \quad C^{\circ}_6 = \{(\mathtt{in}\, m.\, \mathtt{out}\, m.\, M^{\circ}, 1)\} \quad C^{\circ}_7 = \{(\mathtt{out}\, m.\, M^{\circ}, 1)\}$$

State S°_{SYS} reports approximate information about all the derivatives of SYS, including obviously the configurations C°_0, C°_1 and C°_2 describing the initial state of Figure 1. The other configurations are added by performing the abstract transitions. Configurations C°_3 and C°_6 model the unfolding of the recursive processes inside ambients *cell* and *mol*, respectively; they are added by **Rec**$^{\circ}$ steps. In both cases the recursive process, P° or M°, is deleted from the configuration precisely because it has multiplicity one (and thus is it has been consumed). Configurations C°_4, C°_5 and C°_7 are introduced by the execution of rule **In**$^{\circ}$, modelling the movement of one ambient *mol* inside ambient *cell*. Configuration C°_4 describes the situation where ambient *cell* contains exactly one ambient *mol* and in parallel process $\overline{\mathtt{out}}\, m.\, P^{\circ}$; configuration C°_5 shows that any number (including zero) of ambients *mol* may appear at top level, because one has moved somewhereelse; configuration C°_7 shows that exactly one process $\mathtt{out}\, m.\, M^{\circ}$ is running inside *mol*. In configurations C°_4 and C°_7, processes $\overline{\mathtt{in}}\, m.\, \overline{\mathtt{out}}\, m.\, P^{\circ}$ and $\mathtt{in}\, m.\, \mathtt{out}\, m.\, M^{\circ}$, respectively, are deleted because of their multiplicity as explained above. No other configurations are needed to describe the execution of **Out**$^{\circ}$ modeling the movement of ambient *mol* out from ambient *cell*.

This analysis is able to capture the relevant feature of SYS and therefore to establish the property discussed in Example 1: *just one* molecule can reside inside the cell, at the time. In fact, in any configuration of *cell* either there are no occurrences of ambients *mol* or there is exactly one occurrence.

It is worth stressing that different configurations are needed to better approximate the possible contents of ambients, in particular of ambient *cell* (e.g. configurations C°_2, C°_3 and C°_4). This permits to have a very precise information about its possible contents at any computation step, and consequently to substantially restrict the space of possible interactions. More in details, the analysis captures that, when one ambient *mol* resides inside *cell*, it must be the case that the process running in parallel is $\overline{\mathtt{out}}\, m.\, P^{\circ}$; and thus no other ambient is authorized to enter (capability $\overline{\mathtt{in}}\, m$ indeed has been consumed). By using weaker analyses where all the configurations of a given ambient are merged (as in the widening of Section 4 or in analyses in literature [25,24,26,27,17]), it would not possible to derive this information. Moreover, as we have already pointed out, the occurrence counting analysis of [14] cannot prove such a property since it would count the number of ambients *mol* appearing in the whole system.

We conclude by observing that the analysis of the minor modification of SYS, where several ambients *cell* may appear, establishes this property in a similar way (see SYS_1 in Example 3). □

We conclude by briefly discussing the complexity of the analysis. As a measure of complexity we can take the maximal number of iterations of the fixed-point operator in the worst case. This is the maximal number of different configurations we can have in an abstract state, and therefore they are exponential in the size of the abstract process.

4 A Widening Operator

We present here a *widening operator* [11] which formalizes a natural and simple way for further approximating the analysis of Section 3. Widening operators are introduced precisely to speed up convergency of a fix-point computation. In this setting, a simple widening can be defined by a function $\omega : \mathcal{A} \to \mathcal{A}$, reporting an approximation of an abstract state; this means that $\omega(S_1^\circ) = S_2^\circ$ implies $S_1^\circ \subseteq^\circ S_2^\circ$. This function is intended to be applied at each iteration of the fixed-point computation of function $\Psi^\circ_{S^\circ}$ which defines the analysis (see Def. 5).

The widening operator defined below is based on the simple idea to have *at most one* configuration describing the possible content of any abstract label. This means that all the configurations describing a given label are put together. Moreover, in order to be able to completely tune the complexity related to the size of the set of abstract labels, we make the widening operator parametric w.r.t. a partition of $\widehat{\mathcal{L}}^\circ$, the set of abstract labels. Let $\tilde{\mathcal{L}}^\circ = \widehat{\mathcal{L}}^\circ_{/\cong}$, where $\cong$ is the equivalence induced by the chosen partition.

To formalize the widening operator, we use $\cup^c$ to denote the l.u.b. of two configurations (according to the ordering of Def. 3); also, we extend $\cup^c$ to abstract states for performing the merge of all configurations related to equivalent abstract labels.

Definition 6 (Widening). *Consider $\tilde{\mathcal{L}}^\circ$ a partition of $\widehat{\mathcal{L}}^\circ$. We define the widening operator $\omega : \mathcal{A} \to \mathcal{A}$ as*

$$\omega(S^\circ) = \cup^c S^\circ = \{(\Delta^\circ, \cup^c_{j\in\{1,\dots,k\}} C_j) \mid (\Gamma^\circ, C_s) \in S^\circ, \Delta^\circ \cong \Gamma^\circ \Rightarrow \\ \exists l \in \{1,\dots,k\},\ (\Delta^\circ, C_l) = (\Gamma^\circ, C_s)\}$$

Using the widening operator ω we define a new fixed-point operator $\Psi^\omega_{S^\circ} : \mathcal{A} \to \mathcal{A}$, where the widening is applied at each iteration,

$$\Psi^\omega_{S^\circ}(S_1^\circ) = \omega(S_2^\circ) \ \textit{if}\ \Psi^\circ_{S^\circ}(S_1^\circ) = S_2^\circ.$$

In the following, $\mathfrak{S}_{Coll^\omega}[\![P]\!]$ stands for the result of the fixed-point computation of $\Psi^{\omega}_{S^\circ}$. In this way we obtain a new analysis which is *polynomial* w.r.t the cardinality of $\tilde{\mathcal{L}}^\circ$, a completely tunable parameter.

The following example shows that this new analysis still gives interesting results.

Example 5. Consider the system presented in Example 2,

$$\begin{aligned}
&SYS' ::= [M_1]^{mol_1} \mid \ldots \mid [M_1]^{mol_1} \mid [M_2]^{mol_2} \mid \ldots \mid [M_2]^{mol_2} \mid [E]^e \mid \ldots \mid [E]^e\\
&M_1 ::= \texttt{rec}Y.\,\texttt{in}\,m_1.\,(\texttt{out}\,n_1.\,P_1 + \texttt{out}\,q_1.\,Y)\\
&M_2 ::= \texttt{rec}Y.\,\texttt{in}\,m_2.\,(\texttt{out}\,n_2.\,P_2 + \texttt{out}\,q_2.\,Y)\\
&E ::= \texttt{rec}Y.\,\overline{\texttt{in}}\,m_1.\,\overline{\texttt{in}}\,m_2.\,(\overline{\texttt{out}}\,n_2.\,\overline{\texttt{out}}\,n_1.\,Y + \overline{\texttt{out}}\,q_2.\,\overline{\texttt{out}}\,q_1.\,Y)
\end{aligned}$$

Consider the following partition of names $\{\{m_1, q_1, n_1\}, \{m_2, q_2, n_2\}\}$ (ranged over by abstract names s and r respectively) and of labels $\{\{\{mol_1\}\}, \{\{mol_2\}\}, \{\{e\}\}, \{s \in \widehat{\mathcal{L}}^\circ \mid \ cardinality(s) > 1\}\}$ designed for distinguishing mol_1, mol_2 and e. If we apply the analysis of Section 3 we have $\mathfrak{S}_{Coll^\circ}[\![SYS']\!] = S^\circ$ where [2]

$$\begin{aligned}
S^\circ = \{ \ &(\top, \{(mol_1, [1-\omega]), (e, [1-\omega]), (mol_2, [1-\omega])\}),\\
&(\top, \{(mol_1, [0-\omega]), (e, [1-\omega]), (mol_2, [1-\omega])\}),\\
&(\top, \{(mol_1, [0-\omega]), (e, [1-\omega]), (mol_2, [0-\omega])\}),\\
&(mol_1, \{(M_1^\circ, 1)\}), (mol_1, \{(\texttt{in}\,s.\,R_1, 1)\}), (mol_1, \{(R_1, 1)\}), (mol_1, \{(P_1, 1)\}),\\
&(mol_2, \{(M_2^\circ, 1)\}), (mol_2, \{(\texttt{in}\,r.\,R_2, 1)\}), (mol_2, \{(R_2, 1)\}), (mol_2, \{(P_2, 1)\})\\
&(e, \{(E^\circ, 1)\}), (e, \{((\overline{\texttt{in}}\,s.\,\overline{\texttt{in}}\,r.\,R_3, 1)\}), (e, \{((\overline{\texttt{in}}\,r.\,R_3, 1), (mol_1, 1)\}),\\
&(e, \{(R_3, 1), (mol_1, 1), (mol_2, 1)\}), (e, \{((\overline{\texttt{out}}\,s.\,E^\circ), 1), (mol_1, 1)\})\}
\end{aligned}$$

$$\begin{aligned}
&R_1 = (\texttt{out}\,s.\,P_1 + \texttt{out}\,s.\,M_1{}^\circ), R_2 = (\texttt{out}\,r.\,P_2 + \texttt{out}\,r.\,M_2{}^\circ)\\
&R_3 = (\overline{\texttt{out}}\,r.\,\overline{\texttt{out}}\,s.\,E^\circ + \overline{\texttt{out}}\,r.\,\overline{\texttt{out}}\,s.\,E^\circ).
\end{aligned}$$

The analysis establishes that the binding of mol_1 and mol_2 follows a precise order; indeed, there is no configuration for ambient e showing the presence of mol_2 without that of mol_1. The use of different configurations is essential for this very precise prediction, similarly as explained in Example 4.The widening of Definition 6 as well as the existing analysis [25,24,26,27] do not give sufficiently precise information for proving this property.

However, when compared to the existing analyses [25,24,26,27] the widening gives more precise predictions due to the different treatment of continuations in the style of [17]. This causal aspect is very useful in this setting; for instance, it is adequate for analyzing the system SYS' where one of the two molecules has been removed (with the aim of proving that both are necessary). As an example we consider the case where no molecule mol_2 is present, as modelled by the system $SYS'' ::= [M_1]^{mol_1} \mid \ldots \mid [M_1]^{mol_1} \mid [E]^e \mid \ldots \mid [E]^e$ (the symmetric case is analogous). We have $\mathfrak{S}_{Coll^\omega}[\![SYS'']\!] = S_1^\circ$ where

$$\begin{aligned}
S_1^\circ = \{ \ &(\top, \{(mol_1, [0-\omega]), (e, [1-\omega]))\}),\\
&(mol_1, \{(M_1^\circ, [0-\omega]), (\texttt{in}\,s.\,R_1, [0-\omega]), (R_1, [0-\omega])\}),\\
&(e, \{(E^\circ, [0-\omega]), (\overline{\texttt{in}}\,s.\,\overline{\texttt{in}}\,r.\,R_3, [0-\omega]), (\overline{\texttt{in}}\,r.\,R_3, [0-\omega]), (mol_1, [0-\omega])\})\}
\end{aligned}$$

[2] For simplicity we are assuming that $P_i = 0$ for each $i \in \{1, 2\}$.

The widening shows that none of products P_1 and P_2 is released (in particular it reports that process P_1 cannot run at top-level inside ambient mol_1 as instead happens in the abstract state S° result of the analysis of system SYS'). This property cannot be established by applying the existing analyses [25,24,26,27], precisely because these proposals have a far less precise prediction about the local processes of ambients. Specifically, they do not capture that the continuation of capability $\overline{\mathtt{in}}\, r$, inside ambient e, cannot be exercised; and consequently that capability $\overline{\mathtt{out}}\, s$ cannot be consumed by ambient mol_1 to exit from ambient e. □

5 Conclusions and Related Works

Our analysis introduces many novelties w.r.t. the analyses presented in the literature [25,24,26,27]. In particular: (i) it gives very precise information on occurrence counting (which is local in contrast to standard global information [17,14]); (ii) it permits to obtain more detailed information about the processes and ambients which may reside inside an ambient, at any time. This is obtained by adopting different configurations to describe ambients in different stages of evolution and by adapting the treatment of continuations of [17]. As a consequence, the analysis better captures also causality aspects. Causality issues have been considered in a few type systems [1,2,3] for MA or for its variant Safe Ambients [18]. The types of [1,3] describe the possible contents of ambients by means of a sort of traces and probably could give interesting results when applied to biological systems. They however lack occurrence counting information.

Our analysis is rather expensive from a computational point of view w.r.t. the proposals of [25,24,26,27]. In our opinion, this additional complexity is motivated by the need of capturing *quantitative* and *causality* information which are fundamental in the biological systems setting. Examples 1 and 2 (then commented in Sections 3 and 4) demonstrate the relevance of this information and show the limitations of the existing analyses. Moreover, it is worth noting that also the occurrence counting analysis of [14] for MA has an exponential complexity even if it does not report sufficiently precise information for proving the properties of Examples 1 and 2.

Moreover, our approach offers several possibilities for tuning the precision, and therefore to find out the right balance between precision and computational cost. The abstraction is parametric, in the sense that one can choose *which part of the system he is interested in* by defining equivalence classes of ambients labels and names. Further approximations can easily be derived by following the widening approach of abstract interpretation. The polynomial widening of Section 4 is an interesting example, especially because it gives better results with respect to the existing analyses [25,24,26,27]. This is discussed in Section 4 by considering Example 2.

There are several interesting directions for future works. First of all, we intend to implement an abstract interpretation framework for computing analyses of BA. Furthermore, we would like to design new analyses which to take into account the stochastic and temporal aspects. In particular, in this setting, it seems very important to be able to establish *temporal properties* much more general than invariant properties; in particular to observe the possible evolution paths of a biological system. This is motivated by the variety of recent works concerning temporal logics and model checking for biological systems [19,20,8,4].

Acknowledgements. We would like to thank the anonymous referees for their useful comments.

References

1. T. Amtoft. *Causal Type System for Ambient Movements.* Submitted for publication, 2003.
2. T. Amtoft, A. J. Kfoury and S. M. Pericas-Geertsen. *What are Polymorphically-Typed Ambients?* Proceedings of ESOP'01, LNCS 2028, 206-220, Springer Verlag, 2001.
3. T. Amtoft, H. Makholm and J.B. Wells. *PolyA: True Type Polymorphism for Mobile Ambients.* Proc. of TCS'04, 591–604, Kluwer, 2004.
4. R. Barbuti, S. Cataudella, A. Maggiolo-Schettini, P. Milazzo and A.Troina, *A Probabilistic Calculus for Molecular Systems* Proc. of Workshop CS & P, Informatik Berichte 170, Humboldt University press, vol 202–216, 2004.
5. C. Bodei, P. Degano, C. Priami and N. Zannone. *An enhanced cfa for security policies.* Proc. of WITS'03, 2003.
6. L. Cardelli. *Membrane Interactions.* Proc. of BioCONCUR '03, Electronic Notes in Computer Science, 2003.
7. L. Cardelli and A.D. Gordon. *Mobile ambients.* Theoretical Computer Science 240, 177–213, 2000.
8. N. Chabrier and F. Fages. *Symbolic model-checking of biochemical networks.* In Proceedings of the First International Workshop on Computational Methods in Systems Biology, 149-162, 2003.
9. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.
10. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.
11. P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proc. of PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, 1992.
12. P. Degano, F. Levi and C. Bodei. *Safe Ambients: Control Flow Analysis and Security.* Proc. of ASIAN '00, LNCS 1961, 199-214, Springer Verlag, 2000.
13. J. Feret. *Abstract Interpretation-Based Static Analysis of Mobile Ambients.* Proc. of SAS'01, LNCS 2126, 412-430, Springer Verlag, 2001.
14. R. R. Hansen and J. G. Jensen and F. Nielson and H. R.Nielson. *Abstract Interpretation of Mobile Ambients.* Proc. of SAS'99, LNCS 1694, 135-148, Springer-Verlag, 1999.

15. N. Kam, D. Harel, H. Kugler, R. Marelly, A. Pnueli, E.J.A. Hubbard and M.J. Stern. *Formal Modeling of C. elegans Development: A Scenario-Based Approach.* Proc. 1st Int. Workshop on Computational Methods in Systems Biology (ICMSB 2003), LNCS 2602, Springer-Verlag, 4-20, 2003.
16. R. Hofestadt and S.Thelen. *Quantitative modeling of biochemical networks .* Silico Biology, volume1, 39-53, 1998.
17. F. Levi and S. Maffeis. *On Abstract Interpretation of Mobile Ambients.* Information and Computation 188, 179–240, 2004.
18. F. Levi and D. Sangiorgi. *Mobile Safe Ambients.* TOPLAS, 25(1), 1–69. ACM Press, 2003.
19. R. Mardare and C. Priami. *Logical Analysis of Biological Systems.* Fundamenta Informaticae, 64, 271–285, 2005.
20. R. Mardare, O. Vagin, P. Quaglia and C. Priami. *Model Checking Biological Systems described using Ambient Calculus.* Proc. of CMSB'04, LNCS 3082, 85–103, 2004.
21. H.Matsuno, A.Doi, M.Nagasaki and S.Miyano. *Hybrid petri net representation of gene regulatory network.* Pacific Symposium on Biocomputing (5), 338-349, 2000.
22. R. Milner, J. Parrow and D. Walker. *A Calculus of Mobile Processes.* Information and Computation, 100, 1-77, 1992.
23. F. Nielson, H.R. Nielson, R.R. Hansen. *Validating firewalls using flow logics.* Theoretical Computer Science, 283(2), 381-418, 2002.
24. F. Nielson, H.R. Nielson and H. Pilegaard. *Spatial Analysis of BioAmbients.* Proc. of SAS'04, LNCS 3148, pp. 69–83, Springer-Verlag, 2004.
25. F. Nielson, H.R. Nielson, C. Priami and D. Schuch da Rosa. *Control Flow Analysis for BioAmbients.* BioCONCUR'03, Electronic Notes in Computer Science, 2003.
26. F. Nielson, H.R. Nielson, C. Priami and D. Schuch da Rosa. *Static Analysis for Systems Biology.* Proc. of the winter International Symposium on Information and Communication Technologies, 1–6, Trinity College Dublin, 2004.
27. H. Pilegaard, F. Nielson and H.R. Nielson. *Static Analysis of a Model of the LDL Degradation Pathway.* Proc. of CMSB'05, to appear.
28. C.Priami and P. Quaglia. *Beta binders for biological interactions.* Proceedings of Computational Methods in System Biology, CMSB'04, LNBI, to appear.
29. C. Priami, A. Regev, W. Silverman and E. Shapiro. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. Information Processing Letters, 80 (1), 25–31, 2001.
30. A. Regev, E. M. Panina, W. Silverman, L. Cardelli and E. Shapiro. BioAmbients: an Abstraction for Biological Compartments. Theoretical Computer Science, 325, 141–167, 2004.
31. A. Regev, W. Silverman and E. Shapiro. Representation and Simulation of Biochemical Processes using the pi-calculus process algebra. Proc. of the Pacific Symposium on Biocomputing 2001, 6, 459–470, 2001.

A Parametric Model for the Analysis of Mobile Ambients

Dino Distefano

Department of Computer Science,
Queen Mary, University of London
{ddino@dcs.qmul.ac.uk}

Abstract. In this paper we propose a new *parametric abstract finite* model of Mobile Ambients able to express several properties on processes. The model can be used for the analysis of these properties by means of model checking techniques. The precision of the model can be increased by modifying certain numeric parameters increasingly avoiding thereby the occurrences of false counterexamples in the analysis.

1 Introduction

The calculus of Mobile Ambients (MA) is meant to model *wide area* computations. Introduced in [2], MA has as main characteristic to allow active processes to move between different sites.

A wide range of work has been recently carried out on the analysis of mobile ambients [1,8,13,14,18], mostly based on static-analysis techniques and abstract interpretation [7].

In this paper we propose a *parametric finite abstract* model to analyse properties of mobile ambients processes by model checking — as an alternative to static analysis. Our model is based on techniques introduced first in [11]. Such techniques provide a general framework for modelling and verifying systems whose computation involves manipulation of pointer structures. The model we define here is suitable for verifying a wide range of safety properties of systems among which security properties such as *secrecy*. It has the following features: *(i)* It provides a *safe approximation* of the concrete transition system of processes. *(ii)* It models *finitely* (by means of abstraction) processes that are in principle infinite due to replication (i.e., $!P$). *(iii)* The model depends on two (numeric) *parameters* that can be increased to tune its precision in case false counterexamples are returned by the model checking algorithm.

The analysis we propose is based on the following strategy. Our models, called HABA, are special Büchi automata with some typical characteristic of history-dependent automata [17]. HABA are used to represent the behaviour of an ambient process P. Properties of interest are expressed in the temporal logic *NTL* (introduced in [11]) which is interpreted over HABA runs. Then, the model checking algorithm defined in [9,12] can be used to verify the validity of the properties against the model.

The first contribution of our approach w.r.t. existing analyses of MA lays on its ability to deal finitely with replication. The model distinguishes between P and $!P$ at several levels of precision (due to parametricity). Existing techniques can cope with replication only to a limited extent. They are designed only for abstraction $\{0, 1, \omega\}$ (i.e., none, one, many). Our abstraction goes beyond this range by considering a range $\{0, 1, \ldots, M, \omega\}$ with $M > 1$ parameter of the model. Therefore it is able to detect properties of the kind *"a certain number of copies of ambient n is inside ambient m at the same time"*. The second contribution of our approach is that the model introduced here provides a general and completely automated framework for the verification of properties of MA. This means that the model is *not* limited to some specific safety properties (like static analysis techniques). Many temporal properties expressible by NTL-formulae can be automatically checked on the abstract model giving us the possibility to infer safe answer on ambient processes.

Related work. Our model takes inspiration from the following works. The paper [18] proposes an algorithm detecting process firewalls that are not protective. The technique is based on a control flow analysis and does not distinguish between a process P and $!P$. This technique is enhanced in [13] where the precision of the analysis is improved by the use of information about the multiplicity of the number of ambients occurring within another ambient. The distinction is within the range $\{0, 1, \omega\}$. Another refinement of the analysis proposed in [18], for the special case of Safe Ambients [15], is introduced in [8]. However, the analysis proposed — as the one in [18] — *does not* distinguish between different copies of the same ambient. An abstract interpretation framework for MA is proposed in [14]. Based on [13] and [8] the analysis given in this paper considers some information about multiplicity of the ambients and contextual information. Again, based on [13], the paper [1] defines a more accurate analysis for capturing boundary crossing. Also in this work no information on multiplicities is provided.

A parallel stream of work considers model checking for mobile ambients using spatial logics [5] and in particular ambient logic[3]. In [6] the authors identify a fragment of mobile ambients (where replication is replaced by recursion) verifiable by model-checking. For this fragment, a model-checking algorithm for the ambient logic is proposed. The paper [4] introduces a spatial logic for synchronous π-calculus and investigate its power. A model-checking algorithm is then presented for a class of bounded processes. Our contribution stands somehow between these two independent streams of work in that it applies model checking in a static analysis oriented fashion.

Organisation of the paper. This paper is organised as follows: Section 2 reviews some background on the ambient calculus. Section 3 gives an overview of *NTL* and HABA. Section 4 defines an operational semantics for MA using HABA. Section 5 provides some concluding remarks.

Due to space limitation this paper presents the main ideas and results. More details and proofs are reported in the full version of this paper [10].

2 An Overview of Mobile Ambients

We consider the pure *Mobile Ambients* calculus [2] without communication primitives. Let $\mathcal{N}$ be a denumerable set of *names* (ranged over by a, b, n, m). The set of processes over $\mathcal{N}$ is defined according to the following grammar:

$$N ::= \mathsf{in}\, n \;\Big|\; \mathsf{out}\, n \;\Big|\; \mathsf{open}\, n \qquad\qquad \text{(capabilities)}$$

$$P, Q ::= \mathbf{0} \;\Big|\; (\nu n)P \;\Big|\; P|Q \;\Big|\; !P \;\Big|\; n[P] \;\Big|\; N.P \qquad \text{(processes)}$$

For a process P we write $n(P)$ for its set of names. $\mathbf{0}$ does not perform any action. The restriction $(\nu n)P$ creates a new name called n that is private in the scope of P. $P \mid Q$ is the standard parallel composition of processes P and Q. Replication $!P$ represents an arbitrary number of copies of P and it is used to introduce recursion as well as iteration. $n[P]$ represents an ambient with name n enclosing a running process P. Ambients can be arbitrarily nested. Capabilities provide ambients with the possibility to *interact* with other ambients. In particular, $\mathsf{in}\, n$ has the effect to move the ambient that performs it into a sibling ambient called n (if there exists one). Symmetrically, by $\mathsf{out}\, n$, an ambient nested inside n moves outside; $\mathsf{open}\, n$ dissolves an ambient n nested inside the one performing this capability.

The standard semantics of Mobile Ambients is given in [2] on the basis of a structural congruence between processes, denoted by $\equiv$ (see [2]), and a reduction relation $\rightarrow$. Processes are identified up to α-conversion. Moreover, note that: $n[P]|n[Q] \not\equiv n[P|Q]$ that is, multiple copies of an ambient n have distinct identities; and $!(\nu n)P \not\equiv (\nu n)!P$ that is, the replication operator combined with restriction creates an infinite number of new names. The reduction relation $\rightarrow$ is defined by the rules listed in Table 1.

Table 1. Reduction rules for Mobile ambients

$n[\mathsf{in}\, m.P\vert Q]\vert m[R] \rightarrow m[n[P\vert Q]\vert R]$	$\dfrac{P \rightarrow Q}{n[P] \rightarrow n[Q]}$	$\dfrac{P \rightarrow Q}{P\vert R \rightarrow Q\vert R}$	$\dfrac{P \rightarrow Q}{(\nu n)P \rightarrow (\nu n)Q}$
$\mathsf{open}\, n.P\vert n[Q] \rightarrow P\vert Q$	$m[n[\mathsf{out}\, m.P\vert Q]\vert R] \rightarrow n[P\vert Q]\vert m[R]$		$\dfrac{P' \equiv P \quad P \rightarrow Q \quad Q \equiv Q'}{P' \rightarrow Q'}$

3 An Overview on *NTL* and HABA

In this section we summarise the framework for modelling and model checking systems with pointers introduced in [11].

Navigation Temporal Logic. Let LVAR be a countable set of logical variables ranged over by x, y, z, and *Ent* be a countable set of entities ranged over by e, e', e_1 etc. $\bot \notin \mathit{Ent}$ is used to represent "undefined"; we denote $E^{\bot} = E \cup \{\bot\}$ for arbitrary $E \subseteq \mathit{Ent}$. Navigation Temporal Logic (*NTL*) is a linear temporal logic where quantification ranges over logical variables that can denote entities, or may be undefined. The syntax is defined by the grammar:

$$\alpha \ ::= \ nil \ \Big| \ x \ \Big| \ \alpha\!\uparrow \qquad\qquad \text{(navigation expressions)}$$

$$\varPhi \ ::= \ \alpha = \alpha \ \Big| \ \alpha \ \mathsf{new} \ \Big| \ \alpha \leadsto \alpha \ \Big| \ \varPhi \wedge \varPhi \ \Big| \ \neg\varPhi \ \Big| \ \exists\, x.\, \varPhi \ \Big| \ \mathsf{X}\,\varPhi \ \Big| \ \varPhi\,\mathsf{U}\,\varPhi \quad \text{(formulae)}$$

nil denotes the null reference, x denotes the entity that is the value of x (if any), and $\alpha\!\uparrow$ denotes the entity referred to by (the entity denoted by) α (if any). Let $x\!\uparrow^0 = x$ and $x\!\uparrow^{n+1} = (x\!\uparrow^n)\uparrow$ for natural n. The basic proposition α new states that the entity (referred to by) α is fresh, $\alpha = \beta$ states that α and β are aliases, and $\alpha \leadsto \beta$ expresses that (the entity denoted by) β is reachable from (the entity denoted by) α. The boolean connectives, quantification, and the linear temporal connectives X (next) and U (until) have the usual temporal interpretation. We denote $\alpha \neq \beta$ for $\neg(\alpha = \beta)$, $\alpha \not\leadsto \beta$ for $\neg(\alpha \leadsto \beta)$ and $\forall x.\, \varPhi$ for $\neg(\exists\, x.\, \neg\varPhi)$. The other boolean connectives and temporal operators $\Diamond$ (eventually) and $\Box$ (always) are standard [19]. For example, $\Diamond(\exists x.\, x \neq v \ \wedge \ x \leadsto v \ \wedge \ v \leadsto x)$ expresses that eventually v will point to a non-empty cycle.

Formulae are interpreted over infinite sequences of triples, called *allocation sequences*, $(E_0, \mu_0, \mathcal{C}_0)(E_1, \mu_1, \mathcal{C}_1)(E_2, \mu_2, \mathcal{C}_2)\ldots$ where for all $i \geqslant 0$, $E_i \subseteq Ent$ and $\mu_i : E_i^\perp \to E_i^\perp$ such that $\mu_i(\perp) = \perp$; μ_i encodes the pointer structure of E_i. $\mathcal{C}_i$ is a function on E_i such that $\mathcal{C}_i(e) \in \mathbb{M} = \{1, \ldots, M\} \cup \{*\}$ for some fixed constant $M > 0$. The number $\mathcal{C}_i(e)$ is called the *cardinality* of e. Entity e for which $\mathcal{C}_i(e) = m \leqslant M$ represents a chain of m "concrete" entities; if $\mathcal{C}_i(e) = *$, e represents a chain that is longer than M. In the latter case, the entity is called *unbounded.* (Such entities are similar to summary nodes [20], with the specific property that they always abstract from chains.) The cardinality of a set is defined as $\mathcal{C}(\{e_1, \ldots, e_n\}) = \mathcal{C}(e_1) \oplus \ldots \oplus \mathcal{C}(e_n)$ where $n \oplus m = n{+}m$ if $n{+}m \leqslant M$ and $*$ otherwise.

Automata-based models. States in our automata are triples $(E, \mu, \mathcal{C})$, called *configurations.* Let CONF denote the set of all configurations ranged over by γ and γ'. Configurations that represent the same pointer structure at different abstraction levels are related by *morphisms.* For $\gamma, \gamma' \in$ CONF, a morphism is surjective function $h : E_\gamma \to E_{\gamma'}$ which maintains the abstract shape of the pointer dependencies represented by the two related configurations. Moreover, a (pure) chain may be abstracted to a single entity while keeping the cardinality invariant. That is, the cardinality of an entity $e \in \gamma'$ is equal to the sum of the cardinalities of the entities in $h^{-1}(e)$. Collapsing chains to single entities —provided correspondence of the cardinality— is the mechanism used by morphisms to associate to a configuration another more abstract configuration.

Although morphisms provide us with a tool for abstraction of pointer structures, they do not model the dynamic evolution of such structures. To reflect the execution of pointer-manipulating operations as well as the creation or deletion of entities we use *reallocations.* For $\gamma, \gamma' \in$ CONF, a *reallocation* is a multi-set $\lambda : (E^\perp \times E'^\perp) \to \mathbb{M}$ which redistributes (but preserves) cardinalities on E to

E'. More precisely, the total cardinality $\bigoplus_{e' \in E'} \lambda(e, e')$ allocated by λ to $e \in E$ equals $\mathcal{C}(e)$; and the total cardinality $\bigoplus_{e \in E} \lambda(e, e')$ assigned to $e' \in E'$ equals $\mathcal{C}'(e')$. As in the case of morphisms, one entity can be related by a reallocation to more than one entity only if these form a chain. Note that the identity function *id* is a reallocation. We write $\gamma \overset{\lambda}{\rightsquigarrow} \gamma'$ if there is a reallocation (named λ) from γ to γ'. Reallocations are a generalisation of the idea of identity change as present in history-dependent automata [17]: besides the possible change of identity of entities, it allows for the evolution of pointer structures[1].

To model the dynamic evolution of a system manipulating (abstract) linked lists, we use a generalisation of Büchi automata where each state is a configuration and transitions exist between states only if these states can be related by means of a reallocation reflecting the possible change in the pointer structure.

Definition 3.1. *A* high-level allocation Büchi automaton *(HABA) $\mathcal{H}$ is a tuple $\langle X, C, \rightarrow, I, \mathcal{F} \rangle$ with:* (i) $X \subseteq$ LVAR*, a finite set of logical variables;* (ii) $C \subseteq$ CONF*, a set of configurations (also called* states*);* (iii) $\rightarrow \subseteq C \times (\mathit{Ent} \times \mathit{Ent} \times \mathbb{M}) \times C$*, a* transition relation*, s.t. $c \rightarrow_{\lambda} c' \Rightarrow c \overset{\lambda}{\rightsquigarrow} c'$;* (iv) $I : C \rightharpoonup 2^{\mathit{Ent}} \times (X \rightharpoonup \mathit{Ent})$*, an* initialisation *function such that for all c with $I(c) = (N, \theta)$ we have $N \subseteq E$ and $\theta : X \rightharpoonup E$.* (v) $\mathcal{F} \subseteq 2^{C}$ *a set of sets of* accept *states.*

HABA can be used to model the behaviour of systems at different levels of abstraction. In particular, when all entities in any state are concrete (i.e., $\mathcal{C}(e) = 1$ for all e), a concrete model is obtained that is very close to the actual system behaviour.

Model Checking NTL. In [9,12] a model checking algorithm which establishes whether a formula Φ is valid on a given (finite) HABA $\mathcal{H}$ was developed. The model checking algorithm is based on the construction of a tableau graph $G_{\mathcal{H}}(\Phi)$ out of $\mathcal{H}$ and Φ as in [16]. We give here a short summary of this construction.

States of $G_{\mathcal{H}}(\Phi)$ are pairs (q, D) where q is a state of $\mathcal{H}$ and D is the collections of sub-formulae of Φ, and their negations, that possibly hold in q. A transition from (q, D) to (q', D') exists in $G_{\mathcal{H}}(\Phi)$ if $q \rightarrow_{\lambda} q'$ in $\mathcal{H}$ and, moreover, for each sub-formula $\mathsf{X}\,\Psi$ in D there exists a "corresponding" Ψ in D'. Here, the correspondence is defined modulo the reallocation λ. A *fulfilling path* in $G_{\mathcal{H}}(\Phi)$ is then an infinite sequence of transitions — starting from an initial state — that also satisfies all the "until" sub-formulae $\Psi_1 \,\mathsf{U}\, \Psi_2$. That is, if $\Psi_1 \,\mathsf{U}\, \Psi_2$ is in a given state in the sequence, then a corresponding Ψ_2 (modulo a sequence of reallocations) occurs in a later state. Fulfilling path are related with the validity of Φ. More precisely, Φ is valid in $\mathcal{H}$ (written $\mathcal{H} \models \Phi$) iff there does not exist a fulfilling path in $G_{\mathcal{H}}(\neg \Phi)$. The existence of a *self-fulfilling strongly connected sub-component* (SCS) in $G_{\mathcal{H}}(\neg \Phi)$ provides us with a necessary criterion for the existence of a fulfilling path. The tableau graph of a finite HABA is always finite and its number of SCSs is finite as well. Moreover, since the property of

[1] A complete treatment of morphisms and reallocations of pointer structures can be found in [9,11,12].

self-fulfilment is decidable, this gives rise to a mechanical procedure for verifying the validity of formulae.

In [9,12] we also showed that if a formula is valid in an abstract HABA $\mathcal{H}$, then it is valid in all concrete (infinite-state) automata $\mathcal{H}_c$ represented by $\mathcal{H}$. Therefore it is enough to verify the validity on the finite-state abstract automata to infer the validity of the property in all its concretizations. As usual in model checking of infinite-state systems in the presence of abstraction the algorithm is sound but not complete in the sense that it might return *false negatives.* This means that if the algorithm fails to show that Φ is valid in $\mathcal{H}$ then it cannot be concluded that Φ is *not* satisfiable (by some run of $\mathcal{H}$). However, since such a failure is always accompanied by a "prospective" counterexample of Φ, further analysis or testing may be used to come to a more precise conclusion.

4 An Abstract Operational Model for Mobile Ambients

Before defining our model we give two motivating examples.

Example 1. In [8] the following system is considered. Ambient m wants to send a message to ambient b. Messages are delivered enclosed in a wrapper ambient that moves inside the receiver which acquires the information by opening it. For secret messages we want to be sure that they can be opened *only* by the receiver b: $SYS_1 = m[mail[\mathsf{out}\, m.\mathsf{in}\, b.msg[\mathsf{out}\, mail.D]]] \mid b[\mathsf{open}\, msg] \mid \mathsf{open}\, msg$.

Data D is secret, *mail* is the pilot ambient that goes out of m to reach b. The outer-most ambient attempts to access the secret by $\mathsf{open}\, msg$. Once inside b, the wrapper *mail* is opened and b reads the secret D. For the process SYS_1 we want to guarantee that the property (UA): *"no untrusted ambients can access D"* holds.

The previous example illustrates the relevance of *secrecy* in wide-area computations. However, there are other important properties which are relevant for the safety of systems. An instance is given in the following.

Example 2. Let us assume that a distributed network of an organisation (e.g., a bank) has a server used by a certain number of clients to execute critical operations (e.g., buying/selling stocks). A rather trivial implementation could be the following system:

$$\begin{aligned} SYS_2 &= Serv[PORT|PORT|PORT|Exec] \mid Cl[REQ] \mid Cl[REQ] \\ PORT &= P[\mathsf{in}\ Req.\mathsf{in}\ Exec] \\ REQ &= !Req[\mathsf{out}\, Cl.\mathsf{in}\ Serv.\mathsf{open}\ P.DATA] \end{aligned}$$

A client Cl asks the server to execute an order (buying/selling) by sending a request. Details of orders are contained in $DATA$. The ambient Req, implementing a request, leaves the client and goes into the server. Once there, Req uses one of the available $PORT$s which are the data structures used by the server to execute the requests. The port P moves the request to some process $Exec$

which executes the order in *DATA* (and then it gives back *P* to *Serv*). As any other real-life server, the bank's server can accept a limited amount of requests at the same time. The risk is that its finite number of internal data structures (ports *P*s) are consumed in pending requests not yet completed. This can result in an overflow and the server must be rebooted while — in the meanwhile — the bank may be loosing millions. Predicting the number of requests may be difficult. This is mostly because the clients place their orders following some mathematical models which depend from several random variables.

Safety for such kind of systems involves the *number* of requests that the server has to deal with at each time. It is essential that the property no-overflow (NO) holds, i.e. *at any point in time the server has to deal with a number of requests smaller or equal to the size of its data structures (in this case 3 ports).*

Now, suppose that the system is expanded and new clients are added to the bank's network. Therefore the designer of the system decides to implement some strategy meant to avoid overflows. The system is upgraded with a buffer using the following strategy. If the server gets shorter in ports it sends a broadcast message (*BCAST*) to its clients and informs them to address their requests to the buffer (instead that the server). From that moment the server accepts only requests from the buffer which forwards client's orders when the server ask for one (by the ambient *ASK_BUF*). When the server has executed enough requests and its number of free ports get back to normal, the server broadcasts another message (*ADDR_to_Ser*) to the clients to inform them that from that moment on they can again address their requests directly to the server. The designer implements this idea in the following new system:

$$\begin{aligned}
SYS_2 &= SER \mid Cl[REQ|to\_Ser] \mid Cl[REQ|to\_Ser] \\
&\quad \mid Cl[REQ|to\_Ser] \mid Cl[REQ|to\_Ser] \\
REQ &= !Req[\mathsf{in}\ to\_Ser.\mathsf{out}\ to\_Ser.\mathsf{out}\ Cl.\mathsf{in}\ Serv.\mathsf{open}\ P.DATA|inB.DATA] \\
BUF &= Buf[!Req_B[\mathsf{open}\ Ask\_Buf.\mathsf{open}\ B.\mathsf{out}\ Buf.\mathsf{in}\ Serv.\mathsf{open}\ P]] \\
ASK\_BUF &= !Ask\_Buf[\mathsf{open}\ Ready\_for\_req.\mathsf{out}\ Serv.\mathsf{in}\ Buf.\mathsf{in}\ Req_B] \\
BCAST &= BCast[\mathsf{out}\ Serv.\mathsf{in}\ Cl.\mathsf{open}\ to\_Ser.B[\mathsf{open}\ Req.\mathsf{out}\ Cl.\mathsf{in}\ Buf.\mathsf{in}\ Req_B]] \\
ADDR\_to\_Ser &= \mathsf{open}\ Norm\_St.to\_Ser[\mathsf{out}\ Serv.\mathsf{in}\ Cl] \\
SER &= Serv[ASK\_BUF|PORT|PORT|PORT|BCAST|BCAST|BCAST \\
&\quad |BCAST|ADDR\_to\_Ser|ADDR\_to\_Ser|ADDR\_to\_Ser|ADDR\_to\_Ser]
\end{aligned}$$

Now, it should be formally verified that this patch properly avoids any overflows, i.e., in this new system the property no-overflow (NO) holds. Note that (NO) cannot be accurately verified by analyses dealing only with multiplicities $\{0, 1, \omega\}$ as those found in the literature. Other example properties that this system should have and we might wish to verify are: (REQ): *any request eventually reaches the server*; and (REQB): *an ambient Req_B leaves the buffer only after the server has asked for a new request by sending the message Ask_Buf.*

In this paper we are concerned with the verification of the kind of properties described in these two examples.

4.1 HABA Modelling Approach

Due to replication, the concrete transition systems of processes are infinite. Since we want to use model checking as analysis technique for processes it is essential that their representation in the model is finite. A naive encoding of the process topology would be hopeless. Therefore, we focus only on essential information which allow us to infer the properties we need. Along the lines of [8,13,14,18], the information we retrieve from a mobile ambient process P is: *which ambients may end up in which other ambient.* To model P we introduce a classification among the entities in use. For any ambient a occurring in P we have:

– A special entity a^{ho} (called a's *host*) is used to record, at any point in the computation, the ambients (hosted) directly inside *any* copy of ambient a. a^{ho} is fixed, i.e., during the computation its position within the topology of the process does not change.
– A special entity a^{is} (called the *inactive site* of a). It is the repository where the copies of a are placed when this ambient is inactive. Informally speaking, inactive means that a cannot yet execute any action (see Section 4.2). As a^{ho}, also a^{is} does not move during the computation.
– All other entities —distinct from a^{ho} and a^{is}— represent instances of the ambient a. A concrete entity can move according to the capabilities of the particular copy of a it represents. Several instances of a may be represented by a single multiple or unbounded entity.

Example 3. State q_{in} in Figure 4 depicts how process SYS_1 of Example 1 is represented in our model. Outgoing references define the child/parent relation μ. Notation $e{:}n$ says that e denotes an ambient with name n. The host of an ambient, say a, keeps track of the ambients directly contained in *any* copy of a. Thus, ambients m and b are inside the outer-most ambient @, whereas *mail* is inside m. Ambients b and *mail* are empty. Hosts entities are depicted as squares and inactive sites as patterned squares. *msg* is *inactive* since in the beginning it cannot execute any action. Only when both $\mathsf{out}\, m$ and $\mathsf{in}\, b$ have been consumed, *msg* becomes *active*. Inactive ambients are modelled by having the copies pointing to their inactive site. Figure 2 (left) shows the use of the unbounded entity e_2 (depicted as patterned circle) to model more than M copies of the ambient n.

Preliminary notation. We assume the existence of a global function $\mathsf{A} : Ent \to n(P)$ that associates to every entity e a name of the ambient in P represented by e. For $e \in Ent$, we write $e{:}n$ as a shorthand for $\mathsf{A}(e) = n$ and $e{:}n \in E$ as a shorthand for $e \in E \wedge \mathsf{A}(e) = n$.

We consider two fixed disjoint sets of entities: the set of inactive sites $E_P^{\mathsf{is}} = \{n^{\mathsf{is}} \in Ent \mid n \in n(P)\}$ and the set of hosts $E_P^{\mathsf{ho}} = \{n^{\mathsf{ho}} \in Ent \mid n \in n(P)\}$. For every ambient n we assume $\mathsf{A}(n^{\mathsf{is}}) = \mathsf{A}(n^{\mathsf{ho}}) = n$ and in every state of the model n^{is} points to n^{ho}. A HABA state modelling mobile ambients is of the form:

$$q = \langle \gamma, \mathsf{P} \rangle \in \textsc{States}$$

where $\textsc{States} = \textsc{Conf} \times (Ent \rightharpoonup 2^{\mathbf{Proc}})$. The first component $\gamma = (E, \mu, \mathcal{C}) \in \textsc{Conf}$ is a standard HABA configuration as defined in Definition 3.1. Given an entity e, the second component $\mathsf{P} : Ent \rightharpoonup 2^{\mathbf{Proc}}$ associates to e the set of processes e must execute. In figures the component $\mathsf{P}(e)$ is depicted close to e. It is not written if it is the empty process.

Pre-initial state and Initial state. The *pre-initial state* is an artificial state added to the model in order to identify by *NTL*-formulae which ambient an entity represents. The pre-initial state of a process P is constructed in such a way that every entity representing a copy of the ambient n points to the inactive site n^{is}. The structure of the pre-initial state does not reflect the initial topology described by P. *NTL*-formulae exploit the fact that an entity e in the pre-initial state leads to n^{is} to express that e stands for a copy of the ambient n. State q_{pre} in Figure 4 illustrates the pre-initial state of the process SYS_1 of Example 1. Although $\mathsf{A}(e_1) = m$, this information cannot be exploited in *NTL*. However, *NTL*-formulae can refer to the set X of logical variables in the model (see Def. 3.1). By having a variables x_m for any $m \in n(P)$, and by interpreting x_m into m^{is} (see ϑ in Def. 4.2) *NTL*-formulae can refer to m^{is} and therefore to all the other entities. Hence, by the special shape of the pre-initial state, we can use the formula $\exists x : x \rightsquigarrow x_m$ to express that x as a copy of the ambient m.

The *initial-state* models the child/parent relation (i.e. the topology) described by the process in terms of entities and pointers. For example, in Figure 4, q_{in} is the initial state of the process SYS_1 of Example 1. Note that the ambient @ does not have a real instance (it is modelled only by $@^{\mathsf{is}}$ and $@^{\mathsf{ho}}$), therefore we use $@^{\mathsf{is}}$ for the execution of capabilities.

Example 4. The security property (UA) of Example 1 is violated if and only if the following *NTL* formula is satisfied

$$\Phi_{\mathsf{UA}} \equiv \exists x : x \rightsquigarrow x_{msg} \wedge \Diamond(x \not\rightsquigarrow x_{msg} \;\wedge\; x\!\uparrow \neq x_{mail}\!\uparrow \;\wedge\; x\!\uparrow \neq x_b\!\uparrow).$$

Φ_{UA} states that *msg* eventually will be included inside an ambient different from *mail* and b (which are the only trustworthy ones). Note the use of x_{msg}, x_{mail}, x_b to refer to ambient names.

The property no-overflow (NO) (see Example 2) is violated if there are at least four distinct requests inside the server at the same time:

$$\begin{aligned}\Psi_{\mathsf{NO}} \equiv \exists x, y, z, w : x \rightsquigarrow x_{Req} \wedge y \rightsquigarrow x_{Req} \wedge z \rightsquigarrow x_{Req} \wedge w \rightsquigarrow x_{Req} \wedge \\ (x \neq y \wedge x \neq z \wedge x \neq w \wedge y \neq z \wedge y \neq w \wedge z \neq w) \wedge \\ \Diamond(x\!\uparrow = x_{Server} \wedge y\!\uparrow = x_{Server} \wedge z\!\uparrow = x_{Server} \wedge w\!\uparrow = x_{Server})\end{aligned}$$

REQ and REQB (see Example 2) are satisfied if the following formulae hold:

$$\begin{aligned}\Psi_{\mathsf{REQ}} &\equiv \forall x : x \rightsquigarrow x_{Req} \Rightarrow \Diamond(x\!\uparrow = x_{Serv}) \\ \Psi_{\mathsf{REQB}} &\equiv \Box(\forall x{:}x \rightsquigarrow x_{Req_B} \wedge x\!\uparrow = x_{Buf} \Rightarrow (x\!\uparrow \mathsf{U}\, \exists y{:}y \rightsquigarrow x_{Ask\_Buf} \wedge y\!\uparrow = x_{Buf}))\end{aligned}$$

Hence, if $\mathcal{H}_{SYS_1}$ and $\mathcal{H}_{SYS_2}$ are the HABA modelling SYS_1 and SYS_2, the properties are guaranteed to hold if we verify $\mathcal{H}_{SYS_1} \models \neg\Phi_{\mathsf{UA}}$, $\mathcal{H}_{SYS_2} \models \neg\Psi_{NO}$,

$\mathcal{H}_{SYS_2} \models \Psi_{REQ}$ and $\mathcal{H}_{SYS_2} \models \Psi_{REQB}$. That can be automatically checked using the model checking algorithm defined in [9,12].

Canonical form for configurations. As models we use HABA whose configurations are in a special form called canonical. The main advantage of canonical configurations is that the resulting HABA is proved to be finite-state [9]. Informally, given a $L > 0$, a configuration γ is *L-canonical* (or *in L canonical form*) if (a) only concrete entities are closer than $L + 1$ pointer dereferences from a host; and, (b) there are no pure chains longer than $L + 1$. For every configuration γ its canonical form exists and it is unique (denoted by $\mathsf{cf}(\gamma)$). $\mathsf{cf}(\gamma)$ is determined by the unique morphism $h_{\mathsf{cf}} : \gamma \to \mathsf{cf}(\gamma)$.

The Parameters M and L. The precision of automaton $\mathcal{H}$ is ruled by two parameters: L controlling the canonical form; and M defining the minimum number of copies of an ambient represented by a single unbounded entity. Due to canonical form, non-concrete entities are not direct children of hosts: there are L concrete entities in between although all of them represent different instances of the *same* ambient. In other words, a chain of entities $e{:}b, e'{:}b\ldots$ pointing to a host, say a^{ho} represents a set of instances of b inside a. By L and M we are able to distinguish that inside a there are no instances of the ambient b; or there are *precisely* i instances of b with $1 \leq i \leq L + M$; or there are more than $L + M$ instances of b. Since L and M are parameters of the model they can be properly tuned to accomplish a more precise model. For example, assume $M = 1$ and $L = 3$. In q_3 of Figure 3, we know that inside a there are *exactly* two instances of n and *any* number of b's copies strictly greater than 4.[2]

4.2 Coding Processes into HABA Configurations

In this section we define a function $\mathcal{D}$ that codes a given process P into a HABA state. $\mathcal{D}$ returns: *(i)* a configuration γ that models P's topology; and *(ii)* a function P that associates to every entity the set of capabilities. We first define all the auxiliary elements necessary to $\mathcal{D}$'s definition.

Union configuration. For a configuration γ, let $E^{\mathsf{c}}_{\gamma} = E_{\gamma} \backslash (E^{\mathsf{ho}}_{P} \cup E^{\mathsf{is}}_{P})$ be its set of non-fixed entities. For configurations γ, γ' such that $E^{\mathsf{c}}_{\gamma} \cap E^{\mathsf{c}}_{\gamma'} = \varnothing$, the *union configuration* is $\gamma \uplus \gamma' = (E_{\gamma} \cup E_{\gamma'}, \mu, \mathcal{C})$ where: $\mathcal{C}(e) = \mathcal{C}_{\gamma'}(e)$ if $e \in E_{\gamma'} \backslash E_{\gamma}$ and $\mathcal{C}(e) = \mathcal{C}_{\gamma}(e)$ otherwise; and

$$\mu(e) = \begin{cases} \mu_{\gamma}(e) & \text{if } e \in E_{\gamma} \\ \mu_{\gamma'}(e) & \text{if } e \in E_{\gamma'} \backslash E^{\mathsf{c}}_{\gamma'} \\ \mathit{first}(\{e'{:}a \in E^{\mathsf{c}}_{\gamma} \mid b^{\mathsf{ho}} \in \mu^{*}(e')\} \cup \{b^{\mathsf{ho}}\}) & \text{if } e{:}a \in E^{\mathsf{c}}_{\gamma'} \text{ and } \mu_{\gamma'}(e) = b^{\mathsf{ho}} \\ \mu_{\gamma'}(e) & \text{otherwise} \end{cases}$$

[2] Between copies of the same ambient, we draw dashed horizontal arrows to stress that, at the conceptual level, these arrows do not describe a child/parent relation as the solid vertical ones.

For $e{:}a \in E_{\gamma'}$, $\mu(e)$ assigns the first entity in the queue of copies of a. If both configurations have copies of an ambient, say b, inside the same ambient, say a, the union appends the copies of the second configuration to those of the first one. The union for P is defined point-wise: let $q = \langle \gamma, \mathsf{P} \rangle$ and $q' = \langle \gamma', \mathsf{P}' \rangle$ and $e \in E_\gamma \cup E_{\gamma'}$, then

$$(\mathsf{P} \uplus \mathsf{P}')(e) = \begin{cases} \mathsf{P}(e) \cup \mathsf{P}'(e) & \text{if } e \in E_\gamma \cap E_{\gamma'} \\ \mathsf{P}(e) & \text{if } e \in E_\gamma \backslash E_{\gamma'} \\ \mathsf{P}'(e) & \text{if } e \in E_{\gamma'} \backslash E_\gamma \end{cases}$$

Finally the union of states is $\langle \gamma, \mathsf{P} \rangle \uplus \langle \gamma', \mathsf{P}' \rangle = \langle \gamma \uplus \gamma', \mathsf{P} \uplus \mathsf{P}' \rangle$.

Sub-processes executed by ambients. Given a process P the function $\rho : \mathbf{Proc} \rightarrow 2^{\mathbf{Proc}}$ returns the set of sub-processes that the ambient containing P can execute:

$$\begin{array}{lll} \rho(\mathbf{0}) = \varnothing & \rho(M.Q) = \{M.Q\} & \rho(Q \mid Q') = \rho(Q) \cup \rho(Q') \\ \rho(m[Q]) = \varnothing & \rho(!Q) = \{!Q\} & \rho((\nu n)Q) = \rho(Q) \end{array}$$

Processes belonging to nested ambients are not returned. Note that because we do not distinguish between $\nu!P$ and $!\nu P$ we can delete restriction[3].

Enabled and active ambients. An *enabled* ambient is an ambient which is ready to perform some action. Syntactically enabled ambients are those not guarded by a capability. The corresponding semantic notion is being active. In state q, the ambient n *is active* if $\nexists e \in E_{\gamma_q} : \mu_{\gamma_q}(e) = n^{\mathsf{is}}$. If n is not active it is called *inactive*. In the operational model only entities related to active ambients can execute capabilities.

Constructing the state. We use the following abbreviation for a state composed only by two entities.

$$\begin{aligned}(e_1, k_1, P_1) \twoheadrightarrow (e_2, k_2, P_2) = \langle \{e_1, e_2\}, \{e_1 \mapsto e_2\}, \{e_1 \mapsto k_1,\ e_2 \mapsto k_2\}, \\ \{e_1 \mapsto P_1,\ e_2 \mapsto P_2\} \rangle\end{aligned}$$

The next function $\Omega(a, P, k, \mathsf{act})$ returns a HABA state representing the process P contained inside the ambient a. The parameter k deals with cardinalities. The parameter act is a boolean that instructs Ω to construct the configuration with the active or with the inactive representation of its ambients. Formally,

[3] However, we assume that names occurring bound inside restriction are all distinct from each other and from the free names.

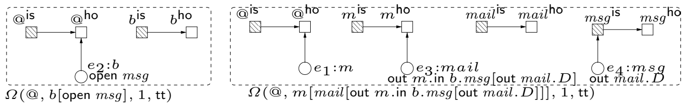


Fig. 1. HABA states returned by $\Omega(@, m[mail[\mathsf{out}\, m.\mathsf{in}\, b.msg[\mathsf{out}\, mail.D]]], 1, \mathsf{tt})$ and $\Omega(@, b[\mathsf{open}\, msg], 1, \mathsf{tt})$

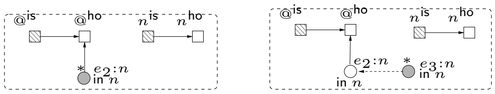


Fig. 2. Left: HABA state returned by $\Omega(@, !n[\mathsf{in}\, n], 1, \mathsf{tt})$. Right: Its 1-canonical form.

$\Omega : \mathcal{N} \times \mathbf{Proc} \times \mathbb{M}^* \times \mathbb{B} \to$ States is given by:

$$\begin{aligned}
\Omega(a, \mathbf{0}, k, \mathsf{act}) &= (a^{\mathsf{is}}, 1, \mathbf{0}) \rightharpoonup (a^{\mathsf{ho}}, 1, \mathbf{0}) \\
\Omega(a, m[Q], k, \mathsf{act}) &= \Omega(a, \mathbf{0}, k, \mathsf{act}) \uplus \Omega(m, Q, k, \mathsf{act}) \\
&\quad \uplus \begin{cases} (e, k, \rho(Q)) \rightharpoonup (a^{\mathsf{ho}}, 1, \mathbf{0}), \text{ if } \mathsf{act} \\ (e, k, \rho(Q)) \rightharpoonup (m^{\mathsf{is}}, 1, \mathbf{0}) \text{ otherwise} \end{cases} \\
&\qquad\qquad \text{where } e{:}m \text{ is fresh} \\
\Omega(a, Q_1|Q_2, k, \mathsf{act}) &= \Omega(a, Q_1, k, \mathsf{act}) \uplus \Omega(a, Q_2, k, \mathsf{act}) \\
\Omega(a, (\nu n)Q, k, \mathsf{act}) &= \Omega(a, Q, k, \mathsf{act}) \\
\Omega(a, !Q, k, \mathsf{act}) &= \Omega(a, Q, *, \mathsf{act}) \\
\Omega(a, N.Q, k, \mathsf{act}) &= \Omega(a, \mathbf{0}, k, \mathsf{act}) \uplus \Omega(a, Q, k, \mathsf{ff})
\end{aligned}$$

The representation of $m[Q]$ in a comprehends $a^{\mathsf{is}}, a^{\mathsf{ho}}$, the sub-state of Q inside m and a configuration with a non fixed entity e standing for the copy of m in a. Depending on the parameter act, this representation can be either the active or the inactive one. $\Omega(a, !Q, k)$ changes the cardinality from k to $*$. Finally, the representation of $N.Q$ inside a has the *inactive* representation for the process Q.

Example 5. Figure 1 shows $\Omega(@, m[mail[\mathsf{out}\, m.\mathsf{in}\, b.msg[\mathsf{out}\, mail.D]]], 1, \mathsf{tt})$ and $\Omega(@, b[\mathsf{open}\, msg], 1.\mathsf{tt})$. In the former, note the different representation between active ambients (@, m, $mail$) and inactive (msg). The left part of Figure 2 shows a state involving replication. We have $\Omega(@, !n[\mathsf{in}\, n], 1, \mathsf{tt}) = \Omega(@, n[\mathsf{in}\, n], *, \mathsf{tt})$ therefore, the entity e_2 modelling the copies of n, becomes unbounded.

Definition 4.1 (Process encoding). *The* process encoding *function* $\mathcal{D}$:**Proc**$\to$ States *is given by* $\mathcal{D}(P) = \langle \mathsf{cf}(\gamma), \mathsf{P}[@^{\mathsf{is}} \mapsto \rho(P)] \rangle$ *where* $\Omega(@, P, 1, tt) = \langle \gamma, \mathsf{P} \rangle$.

The existence of a unique canonical form is proved in [9,12]. The state in the left part of Figure 2 is not L-canonical for any $L > 0$. The canonical form for $L = 1$ is shown on the right side. Note that $\mathcal{D}$ assigns to $@^{\mathsf{is}}$ the set $\rho(P)$ with the capabilities to be executed by @. For any process P, its pre-initial state is given by $q_{pre} = \Omega(@, P, 1, \mathsf{ff})$ and the initial state by $q_{in} = \mathcal{D}(P)$. Figure 4 shows q_{pre} and q_{in} of $SYS1$.

Table 2. Functions for moving ambients used in the operational rules

$act : \mathbf{Proc} \times \mathcal{N} \times \textsc{Conf} \to \textsc{Conf}$ defined by
$act_{Q,a}(\gamma) = (\gamma \backslash \gamma_{\Omega(a,Q,1,\mathsf{ff})}) \uplus \gamma_{\Omega(a,Q,1,\mathsf{tt})}$

$move : \textsc{Conf} \times Ent \times Ent \to \textsc{Conf}$ defined by
$move(\gamma, e, \hat{e}) = (E_\gamma, \mu_\gamma[e \mapsto \hat{e},\ \mu_\gamma^{-1}(e) \mapsto \mu_\gamma(e),\ e' \mapsto e], \mathcal{C}_\gamma)$
where $\mu_\gamma(e') = \hat{e}$, $\mathsf{A}(e') = \mathsf{A}(e)$

$\mathrm{IOUp} : (\textsc{States} \times \mathbf{Proc} \times Ent \times Ent) \to \textsc{States}$ defined by
$\mathrm{IOUp}(q, N.Q, e, \hat{e}) = \langle act_{Q,\mathsf{A}(e)} \circ move(\gamma_q, e, \hat{e}),\quad \mathsf{P}[e \mapsto \mathsf{P}(e) \backslash \{N.Q\} \cup \rho(Q)] \rangle$

$diss : (\textsc{Conf} \times Ent \times Ent) \to \textsc{Conf}$ defined by
$diss(\gamma, a^{\mathsf{ho}}, e{:}b) = (E_\gamma \backslash \{e\},\ \mu_\gamma[\mu_\gamma^{-1}(e) \mapsto a^{\mathsf{ho}},\ \mu_\gamma^{-1}(b^{\mathsf{ho}}) \mapsto a^{\mathsf{ho}}],\ \mathcal{C}_\gamma \restriction E_\gamma \backslash \{e\})$

$\mathrm{OpenUp} : (\textsc{States} \times Ent \times Ent \times \mathbf{Proc}) \to \textsc{States}$ defined by
$\mathrm{OpenUp}(q, N.Q, e'{:}a, e) = \langle act_{Q,a} \circ diss(\gamma_q, a^{\mathsf{ho}}, e), \mathsf{P}[e' \mapsto \mathsf{P}(e') \backslash \{N.Q\} \cup \rho(Q) \cup \mathsf{P}(e)] \rangle$

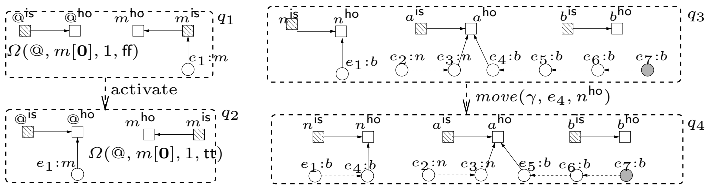


Fig. 3. Left: Rearrangements of pointers performed by $act_{m[\mathbf{0}],@}(\gamma)$. Right: Rearrangements of pointers carried out by $move(\gamma, e_4, n^{\mathsf{ho}})$.

4.3 Configuration Link Manipulations

In our operational model, the computation of a process P corresponds to specific pointer manipulations mimicking the movements of P's ambients (see Figures 4). We will now introduce the functions implementing these pointer manipulations. They will be used in the rules of operational semantics given in Table 3.

State update for in/out. The function $\mathrm{IOUp}(q, N.Q, e, \hat{e})$ in Table 2 performs the overall update of the state q when e moves inside $\hat{e}$ because of the execution of $N \in \{\mathsf{in}\ , \mathsf{out}\ \}$ and continue with Q. There are three kinds of updates to carry out during the execution of N: *(i)* First the pointer rearrangements moving e from its current location to the target location. These updates are performed by $move(\gamma, e, \hat{e})$. *(ii)* Then by applying act, IOUp carries out those rearrangements needed for the activation of the ambients becoming enabled in Q because of the execution of N. *(iii)* Finally, the set of capabilities $\mathsf{P}(e)$ is updated to record that e has executed N and that it must continue with Q. Figure 3 (right part) shows how the configuration changes when e_4 moves inside n^{ho}. In Figure 4, state q_1, in b moves e_3 inside b by making it point to b^{ho}; moreover msg becomes active and it points to $mail^{\mathsf{ho}}$ instead of msg^{is}. Figure 3 (left part) depicts the activation of m by the outer-most ambient @. It corresponds to $act_{m[\mathbf{0}],@}(\gamma)$.

State update for open. The function $\mathrm{OpenUp}(q, N.Q, e'{:}a, e)$ updates the state when $e'{:}a$ executes open of the ambient represented by e. $\mathrm{OpenUp}(q, N.Q, e'{:}a, e)$ performs the following operations: *(i)* It dissolves e using *diss*; *(ii)* It activates the ambients that become enabled; *(iii)* It updates the set of sub-processes that remain to be done by the entity executing open. Note that e' takes the processes $\mathsf{P}(e)$ which were supposed to be executed by e. See the transition between q_3 and q_4 in Figure 4.

4.4 The HABA Semantics of Processes

We can now define HABA $\mathcal{H}_P$ defining the abstract model for process P.

Definition 4.2. *The abstract semantics of a process P is the HABA $\mathcal{H}_P = \langle X_P, S, \rightarrow, I, \mathcal{F}\rangle$ where*

- $X_P = \{x_n \mid n \in n(P)\} \cup \{x_@\}$;
- $S \subseteq$ STATES *such that* $q_{pre}, q_{in} \in S$;
- *let* $\mathcal{R} \subseteq S \times (\mathit{Ent} \times \mathit{Ent} \rightarrow \mathbb{M}) \times S$ *be the smallest relation satisfying the rules in Table 3. Then* $\rightarrow = \mathcal{R} \cup \{(q_{pre}, \lambda_{pre}, q_{in})\} \cup \{(q, id, q) \mid \neg\exists q', \lambda{:}(q, \lambda, q') \in \mathcal{R}\}$;
- $\mathrm{dom}(I) = \{q_{pre}\}$ *and* $I(q_{pre}) = \langle \varnothing, \vartheta\rangle$ *where* $\vartheta(x_n) = n^{\mathsf{is}}$ $(n \in n(P))$.
- $\mathcal{F} = \{\{q \in S \mid (\exists q', \lambda : q \rightarrow_\lambda q') \Rightarrow q = q'\}\}$.

X_P contains a logical variable for each ambient name in P and $x_@$ for the outermost ambient. The transition relation $\rightarrow$ includes a transition from the pre-initial state to the initial state and an "artificial" self-loop for each deadlocked state in $\mathcal{R}$. $\mathcal{F}$ is defined as the set of states whose only outgoing transition is a self-loop. The set I contains only the pre-initial state. The interpretation ϑ allows us to refer to ambient names in *NTL*-formulae (see discussion at page 409).

Operational rules. The execution of a capability $N.Q$, in a given state q, applies the following pattern: γ_q is first modified with the needed link rearrangements into γ'. This is performed by IOUp (for in and out) or OpenUp (for open). Because of the rearrangements of the links, γ' may be not canonical. Therefore, we consider its *safe expansion* $\mathsf{SExp}(\gamma')$[4] and for each of its element γ'' we take the canonical form $\mathsf{cf}(\gamma'')$. The reallocation is defined as: $\lambda = h_{\mathsf{cf}} \circ h^{-1}(\gamma')$ where the morphism h is determined by the safe expansion of γ' and h_{cf} is the morphism giving the canonical form of $h^{-1}(\gamma')$. [12] shows that this is a good definition of reallocation. In q only concrete non-fixed entities modelling an active ambient and directly pointing to a host can move, i.e., $E^{\mathsf{m}} = \{e \in E^{\mathsf{c}}_q \mid \mathsf{A}(e)$ is active, $\mu_q(e) \in E^{\mathsf{ho}}_P\}$. In the rules $E^{\mathsf{m}}_@ = E^{\mathsf{m}} \cup \{@^{\mathsf{is}}\}$. Moreover, *siblings*$(e)$ is the set of ambients having an instance with the same parent of e. *child*(a) returns the entities that are children of the ambient a. *parents*(b) is the set of parents of ambients b. In the **In rule**, if e has $\mathsf{in}\ b.Q$

[4] The safe expansion of a (possibly unsafe) configuration γ' is a finite set of L-safe configurations γ'' which are related to γ' by a morphism (i.e., they represent the same topological structure). Formally: $\mathsf{SExp}(\gamma') = \{\gamma'' \mid \gamma''$ is L-safe and $h : \gamma'' \rightarrow \gamma'\}$. See [12] for an exhaustive treatment.

Table 3. Operational rules for Mobile ambients

In $$\frac{e \in E^{\mathsf{m}}, \quad \mathsf{in}\, b.Q \in \mathsf{P}_q(e), \quad b \in \mathit{siblings}(e)}{q \rightarrow_{\lambda} \mathsf{cf}(\gamma''), \mathsf{P}'}$$

where $\langle \gamma', \mathsf{P}' \rangle = \mathrm{IOUp}(q, \mathsf{in}\, b.Q, e, b^{\mathsf{ho}})$ and $\gamma'' \in \mathsf{SExp}(\gamma')$

Out $$\frac{e \in E^{\mathsf{m}}, \quad \mathsf{out}\, b.Q \in \mathsf{P}(e), \quad \mu(e) = b^{\mathsf{ho}} \quad a \in \mathit{parents}(b)}{q \rightarrow_{\lambda} \mathsf{cf}(\gamma''), \mathsf{P}'}$$

where $\langle \gamma', \mathsf{P}' \rangle = \mathrm{IOUp}(q, \mathsf{out}\, b.Q, e, a^{\mathsf{ho}})$ and $\gamma'' \in \mathsf{SExp}(\gamma')$

Open $$\frac{e \in E^{\mathsf{m}}_{@}, \quad \mathsf{open}\, b.Q \in \mathsf{P}(e), \quad e'{:}b \in \mathit{child}(\mathsf{A}(e))}{q \rightarrow_{\lambda} \mathsf{cf}(\gamma''), \mathsf{P}'}$$

where $\langle \gamma', \mathsf{P}' \rangle = \mathrm{OpenUp}(q, \mathsf{open}\, b.Q, e, e')$ and $\gamma'' \in \mathsf{SExp}(\gamma')$

Bang $$\frac{e \in E^{\mathsf{m}}_{@}, \quad !Q \in \mathsf{P}(e)}{q \rightarrow_{\lambda} \mathsf{cf}(\gamma'), \mathsf{P}'}$$ where $\mathsf{P}' = \mathsf{P}_q[e \mapsto \mathsf{P}_q(e) \cup \rho(Q)]$ and $\gamma' \in \mathsf{SExp}(\mathit{act}_{Q,\mathsf{A}(e)}(\gamma_q))$

and there exists a sibling ambient b then e moves inside b. In the **Out rule**, if e executes $\mathsf{out}\, b.Q$ and its father is ambient b, i.e. $\mu(e) = b^{\mathsf{ho}}$ then e must move in every ambient containing a copy of b. In the **Open rule**, e can execute $\mathsf{open}\, b$, if there exists a $\mathit{child}(\mathsf{A}(e))$ e' modelling a copy of b. Entity e' is dissolved and the component $\mathsf{P}(e)$ acquires the processes contained in $\mathsf{P}(e')$. In the **Bang rule**, if a process $!Q$ is contained in the set of processes that e must execute, then $!Q$ is expanded using the equivalence $!Q \equiv Q|!Q$. Note that we do not need structural rules for parallel composition, restriction, ambients since those constructs are implicitly represented in the configuration of a state.

Example 6. The HABA modelling SYS_1 of Example 1 is depicted in Figure 4. For SYS_1 we want to check the secrecy property (UA) *"no untrusted ambients can access D"* expressed by the *NTL*-formula Φ_{UA} in Example 4. No runs of the HABA satisfies Φ_{UA} therefore in SYS_1 only b can access the secret data D.

Theorem 1. *If $P \rightarrow Q$ then there exists Q' and a finite sequence of $\lambda_1, \lambda_2, \ldots, \lambda_k$ such that $\mathcal{D}(P) \rightarrow_{\lambda_1} \cdots \rightarrow_{\lambda_k} \mathcal{D}(Q')$ and $Q' \equiv Q$.*

This theorem ensures that the HABA semantics of a process P provides a safe approximation of all P behaviours. Although for many processes it provides rather precise information, some limitations occur on processes which combine name restriction and replication. Like other analyses based on static analysis [8,13,14,18], our semantics does not distinguish between processes $!(\nu n)P$ and $(\nu n)!P$. However, our model is able to capture precise information on the number of copies of the same ambients that may be inside another ambient. Therefore it is able to distinguish between P and $!P$. The precision can easily be increased by increasing the parameters L and M.

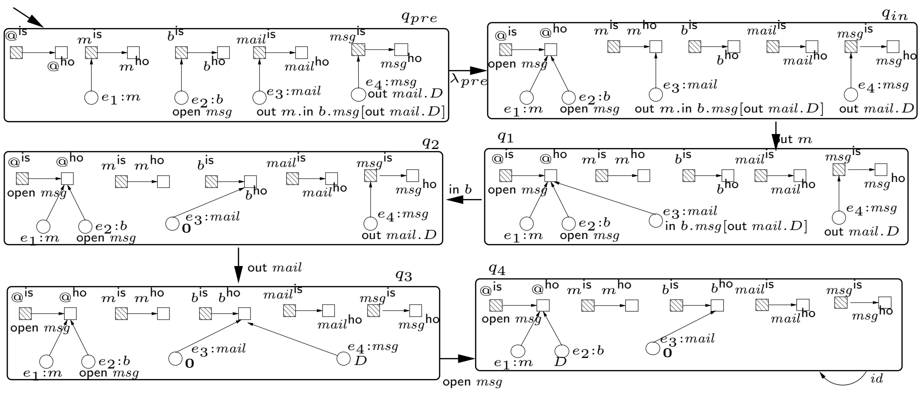


Fig. 4. HABA modelling the system described in Example 1 by the process $SYS_1 = m[mail[\text{out } m.\text{in } b.msg[\text{out } mail.D]]]|b[\text{open } msg]|\text{open } msg$

5 Conclusions

The analysis we have developed in this paper represents an alternative approach which goes beyond the analyses for MA found in the literature. A strong point of our technique seems to rely on its power of counting occurrences of ambients, as well as its flexibility on tuning the precision. Another advantage of our approach w.r.t. static analysis is that the model can be used to investigate properties of the evolution of the computation (via *NTL*). Beside the information "which ambient end up in in which other ambient" our model is able to answer other involved questions which cannot be answered by existing analyses. For example, properties like "it is always the case that whenever a and b are inside n, a exit n before b". Or, "a copy of a does not leave b until another copy of a enters b".

Acknowledgments. I would like to thank Arend Rensink, Joost-Pieter Katoen, Ivana Mijajlović, Josh Berdine, Marco Carbone, Giuseppe Milicia, and Sergio Maffeis for fruitful discussions and helpful suggestions on different stages of this paper. Thanks also to the anonymous referees for detailed comments.

References

1. C. Braghin, A. Cortesi, and R. Focardi. Control flow analysis of mobile ambients with security boundaries. In B. Jacobs and A. Rensink, editors, *FMOODS 2002*, pp. 197–212. Kluwer, 2002.
2. L. Cardelli and A.D. Gordon. Mobile ambients. In *FOSSACS '98*, *LNCS* 1378 pp. 140–155. Springer, 1998.
3. L. Cardelli and A.D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *POLP 2000*, pp. 365–377. ACM Press, 2000.
4. L. Caires. Behavioral and spatial observations in a logic for π-calculus In I. Walukiewicz, editor *FOSSACS 2004*, *LNCS* 2987, pp. 72–89, Springer 2004.
5. L. Caires and L. Cardelli. A spatial logic for concurrency (part I). *Inf. and Comp.*, 186(2):194–235, 2003.

6. W. Charatonik, A.D. Gordon, and J.-M. Talbot. Finite-control mobile ambients. In D. Le Métayer, editor, *ESOP 2002*, *LNCS* 2305, pp. 295–313. Springer, 2002.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL77*, pp. 238–252. ACM, 1977.
8. P. Degano, F. Levi, and C. Bodei. Safe ambients: Control flow analysis and security. In *Asian Computing Science Conference*, *LNCS* 1961, pp. 199–214. Springer, 2000.
9. D. Distefano. On model checking the dynamics of object-based software: a foundational approach. PhD thesis, University of Twente, Nov 2003.
10. D. Distefano. A parametric model for the analysis of mobile ambients. Full version with proofs.
11. D. Distefano, J.-P. Katoen, and A. Rensink. Who is pointing when to whom? on the automated verification of linked list structures. In K. Lodaya, M. Mahajan editors: *FSTTCS 2004*, *LNCS* 3328, pp. 250–262, Springer 2004.
12. D. Distefano, A. Rensink, and J.-P. Katoen. Who is pointing when to whom: on model-checking pointer structures. Tech. Report TR-CTIT-03-12, University of Twente, 2003.
13. R.R. Hansen, J.G. Jensen, F. Nielson, and H.R. Nielson. Abstract interpretation of mobile ambients. In A.Cortesi and G. Filé, editors, *SAS '99*, *LNCS* 1694, pp. 135–148. Springer, 1999.
14. F. Levi and S. Maffeis. An abstract interpretation framework for analysing mobile ambients. In P. Cousot, editor, *SAS 2001*, *LNCS* 2126, pp. 395–411. Springer, 2001.
15. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *POPL 2000*, pp. 352–364. ACM Press, 2000.
16. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL'85*, pp. 97–107. ACM, 1985.
17. U. Montanari and M. Pistore. An introduction to history-dependent automata. In A. Gordon, A. Pitts, and C. Talcott, editors, *HOOTS II*, vol.$\tilde{1}$0 of *ENTCS*. Elsevier Science Publishers, 1997.
18. F. Nielson, H.R. Nielson, R.R. Hansen, and J.G. Jensen. Validating farewalls in mobile ambients. In J.C.M. Baeten and S. Mauw, editors, *CONCUR '99*, *LNCS* 1664, pp. 463–477. Springer, 1999.
19. A. Pnueli. The temporal logic of programs. In *Proceedings FOCS-77*, pp. 46–57. IEEE Computer Society Press, 1977.
20. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *TOPLAS*, 20(1):1–50, 1998.

On the Rôle of Abstract Non-interference in Language-Based Security

Isabella Mastroeni

Department of Computing and Information Sciences - Kansas State University, Manhattan, Kansas, USA
isabellm@cis.ksu.edu

Abstract. In this paper, we illustrate the rôle of the notion of Abstract Non-Interference in language based security, by explaining how it models both the weakening of attackers' observational capability, and the declassification of private information. Namely, we show that in abstract non-interference we model both attackers that can only observe properties of public data, and private properties that can or cannot flow. Moreover, we deepen the understanding of abstract non-interference by comparing it, by means of examples, with some the most interesting approaches to the weakening of non-interference, such as the PER model, robust declassification, delimited release and relaxed non-interference.

Keywords: Language-based Security, Non-Interference, Declassification.

1 Introduction

An important task of language based security is to protect confidentiality of data manipulated by computational systems. Namely, it is important to guarantee that no information, about confidential/private data, can be caught by an external viewer. The standard way used to protect private data is access control: special privileges are required in order to read confidential data. Unfortunately, these methods allow to restrict accesses to data but cannot control propagation of information. Once the information is released from its container, it can be improperly transmitted without any further control. This means that security mechanisms such as digital signature and antivirus scanning, do not provide assurance that confidentiality is maintained during the whole execution of the checked program. This implies that, if a user wishes to keep some data confidential, he might state a policy stipulating that no data visible to other users is affected, within the executed program, by modifying confidential data. This policy allows programs to manipulate and modify private data, as long as visible outputs of those programs do not reveal information about these data. A policy of this sort is called *non-interference* policy [13], since it states that confidential data may *not interfere* with public data, but it is also referred as *secrecy* [25].

The standard approach to non-interference, is based on a characterization of attackers that does not impose any observational or complexity restriction on the attackers' power. This means that the attackers are *all-powerful*, modeled without any limitation in their quest to obtain confidential information. For this reason non-interference, as defined in literature, is an extremely restrictive policy. The problem of refining this kind of security policies has been addressed by many authors as a major challenge in language-based information flow security [22], where refining security policies means weakening standard non-interference checks. In order to adapt security policies to practical cases, we need a weaker notion of non-interference where the power of the attacker (or external viewer) is bounded, and where intentional leakage of information is allowed. In [9], we introduce a weak notion of non-interference for characterizing the secrecy degree of programs by identifying the most powerful attacker that is not able to disclose confidential information by observing the execution of programs, but also in order to characterize the maximal amount of information released. Clearly, this is not the only, and not even the first attempt to weaken the notion of non-interference, from both these points of view, as we will see in Sect. 3. But this is the only one, to the best of our knowledge, that can model, in the same formalism, both weaker attackers and released information. In this paper we show, by means of examples, that different techniques proposed in literature for weakening non-interference (e.g., PER model) and for declassifying information (e.g., selective dependency, delimited release, relaxed non-interference) can be equivalently formalized in abstract non-interference, but moreover we show that in some cases abstract non-interference captures a much more precise notion and allows to derive certifications of the security degree of programs. The aim of this paper is to show how abstract non-interference can be simply adapted in order to cope with different problems concerning secure information flow, and to allow a deep insight of this notion.

2 Abstract Interpretation: A Panoramic View

In the following of this paper we will use the standard framework of abstract interpretation [5,6] for modeling the observational capability of attackers. The idea is that, instead of observing the concrete semantics of programs, namely the concrete values of public data, the attackers can only observe *properties* of public data, namely *abstract semantics* of the program. For this reason we model attackers by means of abstract domains. Abstract domains are used for denoting properties of concrete domains, since their mathematical structure guarantees, for each concrete element, the existence of the *best correct approximation* in the abstract domain. This is due to the property, of abstract domains, of being closed under the concrete greatest lower bound. So for example an abstract domain for the sign analysis is $\mathit{Sign} \overset{\text{def}}{=} \{\mathbb{Z}, 0+, 0-, \{0\}, \varnothing\}$[1], while if we weed out $\{0\}$, then it is no more an abstract domain. Formally, the *lattice of abstract*

[1] $0+ \overset{\text{def}}{=} \{n \mid n \geq 0\}$, $0- \overset{\text{def}}{=} \{n \mid n \leq 0\}$.

interpretations of C is isomorphic to the lattice $uco(C)$ of all the upper closure operators on C [6]. An *upper closure operator* $\rho : C \to C$ on a poset C is monotone, idempotent, and extensive[2]. Closure operators are uniquely determined by the set of their fix-points $\rho(C)$, which are abstract domains formalized independently of the representation of their objects. If C is a complete lattice then $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\ \top, \lambda x.\ x\rangle$ is the lattice of upper closures, where for every $\rho, \eta \in uco(C)$, $\{\rho_i\}_{i\in I} \subseteq uco(C)$ and $x \in C$: $\rho \sqsubseteq \eta$ iff $\eta(C) \subseteq \rho(C)$; $\sqcup_{i\in I}\rho_i = \bigcap_{i\in I}\rho_i$; and $\sqcap_{i\in I}\rho_i = \mathcal{M}(\bigcup_{i\in I}\rho_i)$, where $\mathcal{M}$ is the operation of closing a domain by concrete greatest lower bound, e.g., intersection on power domains.

3 Non-interference and Declassification: The Road Map

Non-interference for programs essentially means that *"a variation of confidential (high or private) input does not cause a variation of public (low) output"* [22]. When this happens, we say that the program has only *secure information flows* [1,4,7,15,25]. This situation has been modeled by considering the denotational (input/output) semantics $[\![P]\!]$ of the program P. In the following, we consider programs where data are typed as private (H) or public (L). Program states, whose set is Σ, are functions (represented as tuples) mapping variables in the set of values $\mathbb{V}$. Any input state s, can be seen as a pair (h, l), where $s^{\mathtt{H}} = h$ is a value for private data and $s^{\mathtt{L}} = l$ is a value for public data. In this case, *(standard) non-interference* can be formulated as follows: Let P be a program

$$\boxed{P \text{ is } \textit{secure} \text{ if } \forall\ s, t \in \Sigma.\ s^{\mathtt{L}} = t^{\mathtt{L}} \ \Rightarrow\ ([\![P]\!](s))^{\mathtt{L}} = ([\![P]\!](t))^{\mathtt{L}}}$$

This notion has been formulated also as a *Partial Equivalence Relation* [14,23].

The limitation of this notion of non-interference is that it is an extremely restrictive policy. Indeed, non-interference requires that *any* change upon confidential data has not to be revealed through the observation of public data. There are at least two problems with this approach. On one side, many real systems are intended to leak some kind of information. On the other side, even if a system satisfies non-interference, too restrictive tests could reject it as insecure. These observations address the problem of *weakening* the notion of non-interference both characterizing the information that is *allowed* to flow, and considering *weaker* attacker models, that can observe only properties of public data. The first problem is treated by means of *declassification* of private information, while the second problem characterizes the observational capability of the attacker.

Different notions of non-interference dealing with declassification have been introduced. The first formalization is *selective dependency* [4], where a property on private input is declassified, by letting the private input range only upon sets of confidential values with the same declassified property. In this case non-interference is checked only whenever a fixed input property holds. So, if we consider the property $\varphi(h) : h\ mod\ 3 = 1$, then non-interference is checked only

[2] $\forall x \in C.\ x \leq_C \rho(x)$.

when the input h satisfies φ. Declassification is considered in presence of active attackers, i.e., attackers that can modify the code of the program in *robust declassification* [26]. Declassification is robust whenever an active attacker cannot disclose more information than what can be observed by a passive attacker. This notion is then better formalized in [19], where also a type system is provided for enforcing robust declassification. More recently two other weakenings of non-interference are provided for dealing with declassification: *delimited release* [21] and *relaxed non-interference* [17]. Both these notions enforce the respective non-interference with a type system, the first on a simple imperative language with explicit declassification of expressions, and the second on λ-calculus. The end-to-end policy in both of them is the same, and it corresponds to selective dependency [4]. In [17] a method is also provided for deriving the declassification policy that makes the program satisfying relaxed non-interference. All these approaches are *qualitative*. There exist also two approaches for *quantifying* the information released [3,18].

As far as the model of the attacker is concerned, a possible limitation of the attackers consists in restricting the possible computational complexity of attackers [8,16], but we can also think of defining non-interference parametric on what the attacker can observe of public data. This model can be given in terms of equivalence relations [14,23,26] or in terms of abstract domains [9,10,11,12].

4 Abstract Non-interference for Modeling Attackers

Consider the program $P \stackrel{\text{def}}{=} l := |l| * \mathit{Sign}(h)$, suppose that $|\cdot|$ is the absolute value function and suppose $\mathit{Sign}(h)$ returns the sign of h, then "*only a portion of l is affected [by h], in this case l's sign. Imagine if an observer could only observe l's absolute value and not l's sign*" [4] then we could say that the program is secure. This is the basic idea in the notion of abstract non-interference [9], used for modeling both weaker attack models and declassification. The idea is that an attacker can observe only some properties, modeled as abstract interpretations of the concrete program semantics. In the following, if $\mathtt{T} \in \{\mathtt{H}, \mathtt{L}\}$, $n = |\{x \in \mathit{Var}(P) | x \text{ of type } \mathtt{T}\}|$, and $v \in \mathbb{V}^n$, we abuse notation by denoting $v \in \mathbb{V}^{\mathtt{T}}$ the fact that v is a possible value for the variables with security type $\mathtt{T}$. The *model of an attacker*, also called *attacker*, is therefore a pair of abstractions $\langle \eta, \rho \rangle$, with $\eta, \rho \in \mathit{uco}(\wp(\mathbb{V}^{\mathtt{L}}))$, representing what it can observe about, respectively, the public input and output of a program. We obtain so far the notion of *narrow (abstract) non-interference* (NANI) denoted $[\eta]P(\rho)$, provided in Table 1. It says that if the attacker is able to observe the property η of public input, and the property ρ of public output, then no information flow concerning the private input is observable from the public output. The problem with this notion is that it introduces *deceptive flows* [9], generated by different *public* output due to different *public* inputs, with the same input property η. Consider, for instance, $[\mathit{Par}]$**if** l **then** $l := 2h + l$ **else** $l := 2h + 1(\mathit{Par})$, we can observe a variation of the output's parity due to the fact that both 0 and 2, for example, are even numbers, revealing a flow not due to different private inputs. Most known mod-

Table 1. Narrow and Abstract Non-Interference (without declassification)

$$[\eta]P(\rho) \text{ if } \forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}, \forall l_1, l_2 \in \mathbb{V}^{\mathtt{L}} \,.\; \eta(l_1) = \eta(l_2) \;\Rightarrow\; \rho(\llbracket P \rrbracket(h_1, l_1)^{\mathtt{L}}) = \rho(\llbracket P \rrbracket(h_2, l_2)^{\mathtt{L}})$$

$$(\eta)P(\rho) \text{ if } \forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}, \forall l \in \mathbb{V}^{\mathtt{L}} \,.\rho(\llbracket P \rrbracket(h_1, \eta(l))^{\mathtt{L}}) = \rho(\llbracket P \rrbracket(h_2, \eta(l))^{\mathtt{L}})$$

els for weakening non-interference (e.g., PER model [23]) and for declassifying information (e.g., robust declassification [26]) corresponds to instances of NANI [9,14]. In order to avoid deceptive interference we introduce a weaker notion of non-interference which considers, as public input, the *set* of all the elements sharing the same property η. Hence, in the previous example, the observable output for l is the set of the possible values obtained by considering all the inputs with the same parity, e.g., if $Par(l) = even$ then we check the parity of $\left\{\, 2h + l \,\middle|\, l \neq 0 \text{ is even} \,\right\} \cup \{2h + 1\}$ which is always unknown, since $2h + 1$ is always odd, and h *does not* interfere with the final parity. This notion, denoted $(\eta)P(\rho)$, is called *abstract non-interference* (ANI) without declassification, and it is formally defined in Table 1. So, if $P \overset{\text{def}}{=} l := |l| * Sign(h)$, and we consider the closure Abs that forgets the sign of an integer number, i.e., $Abs(n) = \{n, -n\}$, then $[Abs]P(Abs)$. While, for example if $P \overset{\text{def}}{=} \textbf{if } h > 0 \textbf{ then } l = m \textbf{ else } l = -m$, then $[id]P(Abs)$. Hence abstract non-interference allows the weakening of attackers models without introducing deceptive interference. Moreover, this abstract model can be used for characterizing the security degree of programs. In particular, the most concrete output observation for a program, given the input one, for both narrow and abstract non-interference can be systematically derived [9]. The idea is that of abstracting in the same object all the elements that, if distinguished, would generate a visible flow. These most concrete harmless attackers, with a fixed η in input, are, respectively, denoted $[\eta]\llbracket P \rrbracket(id)$ and $(\eta)\llbracket P \rrbracket(id)$, both in $uco(\wp(\mathbb{V}^{\mathtt{L}}))$. Hence, in both the programs above, we note that each value n has to be abstracted in $\{n, -n\}$, in order not to generate visible flows, hence the most concrete harmless attacker can at most observe Abs, i.e., $[Abs]\llbracket P \rrbracket(id) = Abs$.

4.1 Abstract Non-interference vs PER Model

In [14], the authors analyze the relationship between abstract non-interference and the PER model of secure information flow. Given the equivalence relations All, Id such that $\forall x, y . x \; All \; y$ and $\forall x, y.\; Id \; y$ iff $x = y$, then the PER model of secure information flow [23] says that P satisfies non-interference if:

$$x \; All \times Id \; y \;\Rightarrow\; \llbracket P \rrbracket(x) \; All \times Id \, f \llbracket P \rrbracket(y)$$

where each state x is partitioned in its confidential and public component, i.e., $x = \langle x^{\mathtt{H}}, x^{\mathtt{L}} \rangle$. In [14] it is shown that this is an instance of NANI, since it considers particular equivalence relations, and since equivalence relations can be modeled by a subset of upper closure operators, called *partitioning* [20]. This also implies that abstract non-interference, without declassification, is strictly more general, as it is proved in [14].

4.2 Abstract Non-interference vs Security for Robust Declassification

In [26], in order to accommodate programs that do leak confidential information, the authors allow the information flow controls to include a notion of *declassified* information, where declassifying means *downgrading* the sensitivity labels on data. Declassification is defined by saying that a *passive attacker*, i.e., attackers that can make only observations of the system and draw inference from those observations, may be able to learn some confidential information by observing the system, but by assumption, that information leakage is allowed by the security policy. The problem is that once a channel is added to the system along which sensitivity labels are downgraded, there is the potential for the channel to be abused to release sensitive information other than that intended. This is clearly possible whenever we are in presence of *active attackers*, i.e., programs running concurrently with the system. When such an attacker cannot obtain more confidential information than what a passive attacker can by simply observing the system, then we say that the system is *robust*. In other words, robust declassification guarantees that if a *passive* attackers may not distinguish between two memories where the secret part is altered, then no *active* attackers may distinguish between these two memories [24].

We compare this paper with ANI without declassification, since robust declassification considers a programming language without explicit declassification and does not characterize the information declassified. It simply derives robustness of declassification by proving a security property that is equivalent to NANI defined on trace semantics [9].

Let $S = \langle \Sigma, \rightarrow \rangle$ be a transition system, and $\langle\!| S |\!\rangle$ the induced trace semantics. Suppose the observational capability of the attacker is characterized by the equivalence relation $\approx$, then the observation of a trace τ of S, through $\approx$ is the trace $\tau/\approx$, such that for each i, $(\tau/\approx)_i \overset{\text{def}}{=} [\tau_i]_\approx$[3], and the observation of S is $\langle\!| S |\!\rangle_\approx \overset{\text{def}}{=} \left\{ \tau/\approx \,\middle|\, \tau \in \langle\!| S |\!\rangle \right\}$. The set of all the traces in S starting from a state σ is denoted $\langle\!| S |\!\rangle(\sigma)$ and two traces are equivalent if they are equal up to stuttering, while two sets of traces are equivalent, i.e., $\equiv$, if all their traces are equivalent: $X \equiv Y \;\Leftrightarrow\; \forall \tau \in X\; \exists \tau' \in Y.\, \tau \equiv \tau'$ and vice versa. The security property for a system S, is $S \models \mathcal{SP}(\approx)$, which holds iff $S[\approx] \supseteq \approx$ (note that $\approx \supseteq S[\approx]$ always holds), where

$$\boxed{\forall \sigma, \sigma' \in \Sigma.\; \sigma\, S[\approx]\, \sigma' \;\Leftrightarrow\; \langle\!| S |\!\rangle(\sigma)_\approx \equiv \langle\!| S |\!\rangle(\sigma')_\approx}$$

Hence, $S \models \mathcal{SP}(\approx)$ holds iff $\forall \sigma, \sigma' \in \Sigma.\; \sigma \approx \sigma' \;\Rightarrow \langle\!| S |\!\rangle(\sigma)_\approx \equiv \langle\!| S |\!\rangle(\sigma')_\approx$, namely the set of the traces starting form σ and observed through $\approx$ is equivalent to the set of the $\approx$-observable traces starting from σ'. A $\approx$-attack is a system $A = \langle \Sigma, \rightarrow_A \rangle$ such that $A \models \mathcal{SP}(\approx)$, which means that the attacker does not know any secret before running with the program. At this point, given an attack A and a system S, both specified in terms of the same set of states Σ, the attack on S by A is the union of the systems: $A \cup S$, and S is *robust* w.r.t. $\approx$, if for

[3] τ_i denotes the i-th element of the trace τ.

all the $\approx$-attacks A, the system $A \cup S$, observed through $\approx$, does not release more information than what is released by S, always observed through $\approx$, i.e., $S[\approx] \subseteq (S \cup A)[\approx]$. Moreover, Theorem 4.1 in [26] says that $S \models \mathcal{SP}(\approx)$ implies the *system is robust* for all the possible $\approx$-attacks.

Consider narrow non interference defined in terms of trace semantics [12], where the abstraction ρ of a trace τ is the trace $\rho(\tau)$ such that $\forall i. \rho(\tau)_i \overset{\text{def}}{=} \rho(\tau_i)$. Consider the attacker modeled by using partitioning closures, i.e., equivalence relations [14], then it is straightforward to prove that narrow non-interference $\models_{\langle\!| S |\!\rangle}$ $[\rho_\approx]S(\rho_\approx)$[4], where $\rho_\approx \overset{\text{def}}{=} \left\{ [x]_\approx \,\middle|\, x \in \mathbb{V}^{\mathtt{L}} \right\} \cup \{\top, \bot\}$, is equivalent to the property $S \models \mathcal{SP}(\approx)$ [9]. This means that, by Theorem 4.1 [26], also NANI can be used for characterizing *robust declassification.* Note that, in both these works there is not explicit declassification in the language, the idea is simply that what is naturally released from the semantics of the system has not to be exploited for obtaining much more information.

Example 1. Consider the following program and its semantics:

$$P \overset{\text{def}}{=} l := 2h + l \text{ and } \langle\!| P |\!\rangle = \left\{ \langle h, l\rangle \to \langle h, 2h + l\rangle \,\middle|\, h \in \mathbb{V}^{\mathtt{H}}, l \in \mathbb{V}^{\mathtt{L}} \right\}$$

This program is neither robust nor satisfies non-interference, being $S[=_L] \subsetneq =_L$ since $\langle h, l\rangle \to \langle h, 2h + l\rangle \not\approx \langle 0, l\rangle \to \langle h, l\rangle$ when $h \neq 0$ and $\approx$ requires the equality of the projection on the low values. On the other hand, if we consider the equivalence relation $\approx_{Par}$[5], then $\forall h_1, h_2.\ \langle h_1, l\rangle \to \langle h_1, 2h_1 + l\rangle \approx_{Par} \langle h_2, l\rangle \to \langle h_2, 2h_2 + l\rangle$, since adding $2h_i$ to l does not change l's parity. Indeed, both $S[\approx_{Par}] = \approx_{Par}$ and narrow abstract non-interference hold. Hence, the program is robust for all the attacks $A \models \mathcal{SP}(\approx_{Par})$. For instance, the attack consisting in the program $A \overset{\text{def}}{=} l := 2l$ cannot observe more than what a passive attacker can, from the execution of P. Indeed,

$$\langle\!| S \cup A |\!\rangle = \{\langle h, l\rangle \to \langle h, 2h + l\rangle \to \langle h, 4h + 2l\rangle, \langle h, l\rangle \to \langle h, 2l\rangle \to \langle h, 2h + 2l\rangle\}$$

and the parity of the public output in all the possible executions does not depend on the input h.

This example shows that, even if declassification in ANI may appear as "vacuously robust" [24] since we do not consider explicitly active attackers, also narrow abstract non-interference can be used for proving *robust* declassification of confidential data, without changing any definition.

5 Abstract Non-interference for Declassification

In abstract non-interference we can also model more *selective* security properties. For example, "*we may not care if output variable b reflects whether input variable a is odd or even. However, we might like to show that b depends upon a in no other way*" [4]. We can also consider another point of view. Suppose that we do want the output variable b does not reflect whether input variable a is odd or even, but we do not care that b depends upon a in other ways.These are two

[4] $\models_{\langle\!| P |\!\rangle}$ $[\eta]P(\rho)$ denotes when NANI is checked by using the trace semantics $\langle\!| P |\!\rangle$.
[5] $\langle h_1, l_1\rangle \approx_{Par} \langle h_2, l_2\rangle$ iff $Par(l_1) = Par(l_2)$.

Table 2. Abstract Non-Interference

$(\eta)P(\phi \rightsquigarrow\!\!\| \rho)$ if $\forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}, \forall l \in \mathbb{V}^{\mathtt{L}} \,.\, \rho([\![P]\!](\phi(h_1), \eta(l))^{\mathtt{L}}) = \rho([\![P]\!](\phi(h_2), \eta(l))^{\mathtt{L}})$
$(\eta)P(\phi \Rightarrow \rho)$ if $\forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}, \forall l \in \mathbb{V}^{\mathtt{L}}\, \phi(h_1) = \phi(h_2) \;\Rightarrow\; \rho([\![P]\!](h_1, \eta(l))^{\mathtt{L}}) = \rho([\![P]\!](h_2, \eta(l))^{\mathtt{L}})$

aspects of declassification: the first specifies what is admitted to flow, the other specifies what cannot flow, admitting that something else may be released.

Abstract non-interference considers a confidential data property, modeled by an upper closure operator ϕ, which represents the confidential property we are interested in keeping secret. This notion is provided in the first definition in Table 2, where $(\eta)P(\phi \rightsquigarrow\!\!\| \rho)$ means that the program P keeps secret ϕ when the attacker is modeled by the I/O pair of observable properties $\langle \eta, \rho \rangle$ and is called *declassified ANI via blocking*. So for example the property $(id)l := l{*}h^2(Sign \rightsquigarrow\!\!\| Sign)$ is satisfied, since the public result's sign does not depend on the private input sign, which is kept secret. In this case, whenever a flow of information is revealed this can only be due to the change of the property ϕ. In this way we model a notion of non-interference which is parametric both on what the attacker can observe of public data and on the confidential property that we want to keep secret.

On the other hand, in ANI [9], we also provide a construction for characterizing the maximal amount of confidential information that *flows*, for a fixed attacker model. This corresponds to characterizing the most concrete property that has to be declassified in order to make the program secure [11]. Declassification can be made explicit also in abstract non-interference, following the idea of selective dependency [4]. Namely, given the property ϕ that we want to declassify, we check abstract non-interference *only* when the private input has the same property ϕ. This notion is provided in the second definition in Table 2, where $(\eta)P(\phi \Rightarrow \rho)$ means that the program P is secure when we let ϕ to flow in presence of an attacker modeled by $\langle \eta, \rho \rangle$, and is called *declassified ANI via allowing*. So for example, consider $P \overset{\text{def}}{=} l := l + (h \ mod\ 3)$, if we want to understand which property flows, we have to characterize which values of the private input generate different public outputs, and we can note that all the elements in $\{\, h \mid h \ mod\ 3 = 0 \,\}$ generate the same output l, such as the elements in $\{\, h \mid h \ mod\ 3 = 1 \,\}$ and in $\{\, h \mid h \ mod\ 3 = 2 \,\}$. In particular, this partition is such that each pair of values in the same set generates the same public output, while each pair of values from different sets generates different public values. This is exactly the *maximal amount of information disclosed*.

The basic idea of the construction is to collect, for each possible public input, all the private inputs that give the same result as public output. In this way we obtain a partition $\Pi(\eta, \rho)_{|_L}$ for each possible property L in input:

$$\begin{aligned} \Pi(\eta, \rho) &\overset{\text{def}}{=} \{\, \langle \{\, h \in \mathbb{V}^{\mathtt{H}} \mid \rho([\![P]\!](h, \eta(l))^{\mathtt{L}}) = A \,\}, \eta(l) \rangle \mid l \in \mathbb{V}^{\mathtt{L}}, A \in \rho \,\} \\ \Pi(\eta, \rho)_{|_L} &\overset{\text{def}}{=} \{\, H \mid \langle H, L \rangle \in \Pi(\eta, \rho) \,\} \end{aligned}$$

So, like delimited release [21], in presence of passive attacker, it is sufficient to declassify the property depending on the public input we are observing. Hence $\forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}$, where $\forall L \in \eta . \pi_L \overset{\text{def}}{=} \Pi(\eta, \rho)_{|_L}$, we have

$$[h_1]_{\pi_L} = [h_2]_{\pi_L} \ \Rightarrow\ \rho(\llbracket P \rrbracket (h_1, L)^{\mathtt{L}}) = \rho(\llbracket P \rrbracket (h_2, L)^{\mathtt{L}}) \tag{1}$$

Let us denote this policy as $\forall L \in \eta . \, (\eta)P(\pi_L \Rightarrow \rho)$. It is clear that checking Eq. 1, could be considered sufficient in presence of only passive attackers, since the standard idea of non-interference compares only the outputs obtained from computations with a fixed public input. But is it a realistic notion of non-interference? Namely, if we have an attacker that observes a system, then we cannot avoid him to observe computations due also to different public inputs, but if this happens then the definition in Eq. 1 is no more a non-interference property. So, if $P \overset{\text{def}}{=}$ **if** $h = l$ **then** $l := 0$ **else** $l := 1$, then the declassified information should be $\Pi(id, id)_l = \{\{l\}, \{\, h \,|\, h \neq l \,\}, \mathbb{Z}, \varnothing\}$. But if the attacker can control the public input, or can observe computations of P with different public inputs for l, then it could collect the results, deducing more than what is declassified about h. For this reason we are interested in a unique property to declassify, which is independent of the fixed public input, namely we derive the most abstract property containing all the properties that should be declassified for each public input and such that $(\eta)P(\phi \Rightarrow \rho)$ [9]:

$$\phi \overset{\text{def}}{=} \bigsqcap_{L \in \eta} \mathcal{M}(\Pi(\eta, \rho)_{|_L}) \tag{2}$$

5.1 Abstract Non-interference vs Enforcing Robust Declassification

In [19] the notion of robust declassification [26] is expressed in a language-based setting and is generalized in order to make untrusted code and data explicitly part of the system rather than appearing only when the attacker is active. In this work the skills of the attacker are made explicit distinguishing between what it can *observe* and what it can *modify*. Hence, each variable on the program has two security classifications, the first says who can observe and the second says who can modify it, e.g., $x_{\mathtt{LH}}$ means that the variable can be observed but cannot be modified by a low level user.

More precisely, in [19], the authors consider a syntax, for defining programs, which includes explicit declassification of expressions to the low security level, and which allows to leave *holes* in the code:

$$\begin{aligned} &e ::= v \mid x \mid e_1\, op\, e_2 \mid \mathit{declassify}(e) \\ &c[\overrightarrow{\bullet}] ::= \mathbf{nil} \mid x := e \mid \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ e\ \mathbf{do}\ c \mid [\bullet] \end{aligned}$$

In these holes the attacker can insert his own code, possibly modifying the computation, i.e., $c[a]$ is the program c under the attack a which means that each hole $[\bullet]$ in c is substituted by the code a. The system $c[\overrightarrow{\bullet}]$ is robust if $\forall \sigma, \sigma' \in \Sigma$ (possible initial memories) and $\forall a, a'$, when both $\langle\!| c[a] |\!\rangle(\sigma)$ and $\langle\!| c[a] |\!\rangle(\sigma')$ terminate, and $\langle\!| c[a] |\!\rangle(\sigma) \approx \langle\!| c[a] |\!\rangle(\sigma')$ then $\langle\!| c[a'] |\!\rangle(\sigma) \approx \langle\!| c[a'] |\!\rangle(\sigma')$, where $\approx$ requires the equality of low-level data. This definition says that a system is robust if any attack cannot change the observable computations of the system.

By considering this view of the problem, we may conclude that it is implicit in non-interference, and therefore also in abstract non-interference, that all the secure systems, where only the input public data can be modified by an untrusted user, are robust. Namely, if we consider the assumption that the attacker may be able to change inputs, i.e., *controls* the inputs, but cannot change the code itself [2], then declassification in ANI is *precisely* robust, as shown in the following examples. Clearly it is a limitation on the possible attacks, and indeed the extension of ANI to all the possible (active) attacks described in [19] deserves further investigation. On the other hand, declassified ANI via allowing provides the construction for deriving *what* is released by a program.

Example 2. 1. Consider $P[\overrightarrow{\bullet}] \overset{\text{def}}{=} [\bullet]; l := l + \mathit{declassify}(h \ \mathit{mod}\ 3)$, where $l : \mathtt{LL}$ and $h : \mathtt{HH}$. This program satisfies robust declassification. On the other hand, we can derive the maximal amount of information disclosed from the program $P' \overset{\text{def}}{=} l := l + h \ \mathit{mod}\ 3$: $\phi = \{\top, 3\mathbb{Z}, 3\mathbb{Z}+1, 3\mathbb{Z}+2, \bot\}$, i.e., it is the most concrete property that has to be declassified in order to guarantee secrecy. We can prove that $(\mathit{id})P(\phi \Rightarrow \mathit{id})$, namely for each possible observation, when we declassify ϕ, the program is secure. This also prove that the program $P[\overrightarrow{\bullet}]$ is robust, since ϕ describes exactly the property $h \ \mathit{mod}\ 3$ declassified, and non-interference cannot be violated even if the attacker controls the input l.

2. Let $P[\overrightarrow{\bullet}] \overset{\text{def}}{=} [\bullet]; \textbf{if } \mathit{declassify}(h = l) \textbf{ then } h := h - l;\ \textbf{else } l := l + h$, where $l : \mathtt{LL}$ and $h : \mathtt{HH}$. In this case the program is not robust since the attacker can modify the input of l and therefore can obtain the value of h. Indeed, if we consider ANI for the program $P' \overset{\text{def}}{=} \textbf{if } h = l \textbf{ then } h := h - l;\ \textbf{else } l := l + h$, and we derive the most concrete property that has to be declassified in order to guarantee secrecy, we obtain the identity:

$$\Pi(\mathit{id}, \mathit{id})_{|_l} = \left\{ \mathbb{Z}, \{l\}, \left\{ h \,\middle|\, h \neq l \right\}, \varnothing \right\} \text{ and } \phi = \bigsqcap_{l \in \mathbb{Z}} \Pi(\mathit{id}, \mathit{id})_{|_l} = \mathit{id}$$

Namely, we proved that the program cannot be made secure unless we declassify the value of private data.

5.2 Abstract Non-interference vs Delimited Release

The aim of delimited release [21] is to find a definition of non-interference which takes into account explicit declassifications in the code of programs. However, explicit declassification does not concern with what is actually leaked during the execution of the program, but simply determines the downgrading policy to check. Delimited release is a generalization of Cohen's selective dependency [4]. In standard non interference we check the security property for all the possible private inputs, this means that the private input can range over the whole domain of possible values. In delimited release the idea is to select, by using the declassification explicitly given (consider the simplified *declassify* operation used in the previous section), for which private inputs we have to check non-interference. Let $\approx_e$ the equivalence relation induced by the evaluation of the expression e, i.e., $\sigma \approx_e \sigma'$ iff $[\![e]\!](\sigma) = [\![e]\!](\sigma')$, we can define $\sigma \approx_E \sigma'$ iff $\forall e \in E. \sigma \approx_e \sigma'$. Delimited release is defined as follows: Let E be the set of all the declassified expressions in the program P

$$\boxed{\forall \sigma, \sigma' \in \Sigma.(\sigma^{\mathtt{L}} = \sigma'^{\mathtt{L}}) \ \wedge \ \sigma \approx_E \sigma' \ \Rightarrow \ [\![P]\!](\sigma)^{\mathtt{L}} = [\![P]\!](\sigma')^{\mathtt{L}}}$$

This generalizes selective dependency because $\approx_E$ is applied to the whole state and not only to the high component. But, whenever the expressions in E depend only on the high input, then the two notions collapse, corresponding also to declassified ANI via allowing. However, whenever the expression E depends also on public inputs, then we obtain a notion of declassified non-interference which depends on the fixed public input, as in Eq. 1, inheriting all the problems expressed for Eq. 1.

At this point, let us see which analogies can be found between declassified ANI and delimited release. In the following, we say that a closure ϕ models the set of declassifications E if $\approx_E \subseteq \approx_\phi$, where $\sigma \approx_\phi \sigma_1$ iff $\phi(\sigma^{\mathtt{H}}) = \phi(\sigma_1^{\mathtt{H}})$.

Example 3. 1. Consider $P \stackrel{\text{def}}{=} avg := \mathit{declassify}((h_1 + h_2 + \ldots + h_n)/n)$. This program satisfies delimited release [21], and if we derive the maximal amount of information disclosed in the program $P' \stackrel{\text{def}}{=} avg = (h_1 + h_2 + \ldots + h_n)/n$, we obtain the following closure:

$$\phi = \left\{ \ \left\{ \ \langle h_1, \ldots, h_n \rangle \,\middle|\, h_1 + \ldots + h_n = x \ \right\} \,\middle|\, x \in \mathbb{Z} \ \right\} \cup \{\mathbb{Z}, \varnothing\}$$

which models exactly the declassified property (being n a known constant).

2. Consider the program

$$P \stackrel{\text{def}}{=} \begin{bmatrix} h_1 := h_1; \ h_2 := h_1; \ \ldots h_n := h_1; \\ avg := \mathit{declassify}((h_1 + h_2 + \ldots + h_n)/n) \end{bmatrix}$$

This program does not satisfy delimited release [21], and if we consider ANI and we derive the maximal amount of information disclosed in the program without the *declassify* operation, we obtain:

$$\phi = \left\{ \ \left\{ \ \langle h, k_2, \ldots, k_n \rangle \,\middle|\, \forall i. k_i \in \mathbb{V}^{\mathtt{H}} \ \right\} \,\middle|\, h \in \mathbb{Z} \ \right\} \cup \{\mathbb{Z}, \varnothing\}$$

namely the maximal amount of information disclosed is the identity on the first private input. Hence the program is not secure, since ϕ does not model the information explicitly declassified, e.g., $\langle 1, 0, \ldots, 0 \rangle \approx_e \langle 0, 1, \ldots, 0 \rangle$ while $\phi(\langle 1, 0, \ldots, 0 \rangle) \neq \phi(\langle 0, 1, \ldots, 0 \rangle)$.

3. Let $P \stackrel{\text{def}}{=}$ **if** $\mathit{declassify}(h \geq k)$ **then** $(h := h - k; l := l + k)$ **else nil**. This program does satisfy delimited release [21]. Consider now abstract non-interference, the maximal information disclosed, for each public input is:

$$\Pi(id, id)_{|_k} = \{\{ \ h \,|\, h \geq k \ \}, \{ \ h \,|\, h < k \ \}, \mathbb{Z}, \varnothing\}$$

Therefore, if we consider the notion of abstract non-interference in Eq. 1, we have that the program is secure, but if we suppose that the attacker can control the public input, or can observe computations corresponding to different public inputs, then we have to compute the maximal amount of information disclosed, that in this case is $\phi = id$. Namely, the program cannot be made secure, unless we declassify the value of private data. For these reasons, when we consider the program under an attack that can change the value of k during the computation, as happens in the following program [21], then it does no more satisfy delimited release.

$$P \overset{\text{def}}{=} \left[\begin{array}{l} l := 0; \\ \textbf{while } n \geq 0 \textbf{ do} \\ \quad k := 2^{n+1}; \\ \quad \textbf{if } \mathit{declassify}(h \geq k) \textbf{ then } (h := h - k; l := l + k) \textbf{ else nil}; \\ \quad n := n - 1; \end{array}\right.$$

4. Consider the program

$$P \overset{\text{def}}{=} \left[\begin{array}{l} h := h \ \mathit{mod}\ 2; \\ \textbf{if } \mathit{declassify}(h = 0) \textbf{ then } (l := 0; h := 0) \textbf{ else } (l := 1; h := 1); \end{array}\right.$$

This program does not satisfy delimited release [21]. The maximal amount of information disclosed in ANI is

$$\phi = \{\{\ h \,|\, h \ \mathit{mod}\ 2 = 0\ \}, \{\ h \,|\, h \ \mathit{mod}\ 2 = 1\ \}, \mathbb{Z}, \varnothing\} \equiv \mathit{Par}$$

The program is not secure, since the amount of information declassified is less than the maximal amount of information released, e.g., $\langle 2, l\rangle \approx_e \langle 3, l\rangle$ (for each l) while $\phi(2) \neq \phi(3)$, i.e, $\langle 2, l\rangle \not\approx_\phi \langle 3, l\rangle$. The secure program is [21]: **if** $\mathit{declassify}(h \ \mathit{mod}\ 2)$ **then** $(l := 0; h := 0)$ **else** $(l := 1; h := 1)$.

These examples show that delimited release captures a notion of non-interference which is weaker than declassified ANI, in the sense that it corresponds to a notion of non-interference (Eq. 1) that holds only in presence of passive attackers unable to collect results due to different public inputs.

5.3 Abstract Non-interference vs Relaxed Non-interference

The notion of delimited release for λ-calculus is *relaxed non-interference* [17]. In this notion a λ-term t, cleaned from the security labels, is secure when it is equivalent to, i.e., can be rewritten as $f(n_1\sigma_1)\dots(n_k\sigma_k)$, where f is a λ-term without any secret variable, σ_i are all secret variables and n_i are downgrading policies (λ-terms) for σ_i such that, for each i, $n_i\sigma_i$ is a public information on σ_i. The term $f(n_1\sigma_1)\dots(n_k\sigma_k)$ satisfies non-interference since, when we fix the public input, included all the $n_i\sigma_i$, we cannot observe any difference in the public output, allowing the confidential information to be leaked in a controlled way, i.e., controlled by the policies n_i. Unfortunately, this definition has some limitations, in particular, even if the authors do not prove that this is the only failure situation, they note that this notion fails in presence of, for example, an exponentially long time attack. However, this failure case is due to the same problems arisen for Eq. 1. The interesting aspect of this paper is that they can derive, from the program, the right downgrading policy that makes the program secure. We show in the following example that this corresponds precisely to the maximal amount of information disclosed for a fixed input [9]. Note that, even if at a glance relaxed non-interference could seem to be a declassification more via blocking than via allowing, this is not true since declassification via blocking is characterized by checking non-interference for the semantics *collecting* the results for all the private inputs that have the same property. Instead, declassification via allowing is characterized by checking non-interference on the concrete semantics but *only* for those private inputs with a fixed property. This is exactly what happens in relaxed non-interference since the inputs $n_i\sigma_i$ are considered public (even if the σ_i's are private), and therefore fixed, hence only the σ_i's that give the same result $n_i\sigma_i$ are considered.

Example 4. Consider the program P [17] with $sec, x, y : \mathtt{H}$, and $in, out : \mathtt{L}$:

$$P \overset{\text{def}}{=} \left[\begin{array}{l} x := hash(sec); y := x \ mod\ 2^{64}; \\ \textbf{if}\ y = in\ \textbf{then}\ out := 1\ \textbf{else}\ out := 0; \end{array}\right.$$

where *hash* is a function. The downgrading policy for *sec* in the corresponding λ-program, is $\lambda sec.\lambda in.(hash(sec)\ mod\ 2^{64}) = in$ [17]. Consider now the maximal amount of information disclosed for each public input value *in*:

$$\begin{array}{rl} \Pi(id, id)_{in} & = \left\{ \left\{ sec \,\middle|\, [\![P]\!](sec, in)^{out} = k \right\} \,\middle|\, k \in \{0,1\} \right\} \\ & = \left\{ \begin{array}{l} \left\{ sec \,\middle|\, (hash(sec)\ mod\ 2^{64}) = in \right\}, \\ \left\{ sec \,\middle|\, (hash(sec)\ mod\ 2^{64}) \neq in \right\} \end{array} \right\} \end{array}$$

where $[\![P]\!](sec, in)^{out}$ is the value resulting in the variable *out*, when the inputs are *sec* and *in*. Note that the closure $\phi_{in} \overset{\text{def}}{=} \Pi(id, id)_{in} \cup \{\top, \bot\}$ corresponds, to the downgrading policy above. Moreover, in ANI we can derive the maximal amount of information disclosed, independently of the public input $\phi = \bigsqcap_{in \in \mathbb{V}^{\mathtt{L}}} \Pi(id, id)_{in}$, making the program also robust in presence of active attackers.

Note that also relaxed non-interference inherits the problems expressed for Eq. 1, since it depends on the particular fixed public input. This means that relaxed non-interference requires the attacker not to be able to observe computations due to different public inputs and to control the public input.

6 Discussion

In [24] the authors provide a road map for classifying all the different approaches to declassification, where declassification means weakening non-interference.

Table 3. Outline

Who	PER model $\equiv [\mathit{Id}]P(\mathit{Id})$
	$S \models \mathcal{SP}(\approx) \equiv \models_{\langle\!\langle P \rangle\!\rangle} [\rho_{\approx}]S(\rho_{\approx})$
What (via Allowing)	Selective Dependency $\equiv (\mathit{Id})P(\phi \Rightarrow \mathit{Id})$
	Delimited Release $\equiv \forall l \in \mathbb{V}^{\mathtt{L}}.\ (\mathit{Id})P(\pi_l \Rightarrow \mathit{Id})$
	Relaxed Non-interference $\equiv \forall l \in \mathbb{V}^{\mathtt{L}}.\ (\mathit{Id})P(\pi_l \Rightarrow \mathit{Id})$
What (via Blocking)	$(\eta)P(\phi \rightsquigarrow\!\!\| \rho)$

They distinguish among four categories, depending on *what* is released, *who* controls the information released, *where* the information is released and *when* the information is released. They introduce abstract non-interference as a particular case of the PER model of information, while in [14] it is proved that, vice versa, the PER model can be seen as a particular case of abstract non-interference. Moreover, abstract non-interference is only in the *what* classification [24], since only its *partial release* aspect is considered, but as we show in this paper, there is also a *who* function that can be treated in terms of abstract non-interference, which is made even more evident by the relation existing between abstract non-interference analysis and robust declassification.

In this paper, we show that our point of view is quite different, in particular, as regards the *who* and *what* distinction. Hence, for us the *who* class captures all the weakenings of non-interference where the notion is weakened by modeling the power of *who* is observing. For this reason, in this class we put the PER model [23], the security policy for robust declassification [26], abstract non-interference [9], and the complexity-based approaches [8,16]. While the *what* class contains all the weakenings consisting in modeling *what* can or cannot flow, independently of who controls information release, and therefore it contains selective dependency [4], enforced robust declassification [19], abstract non-interference [9], delimited release [21], relaxed non-interference [17], and information theory-based approaches [3,18]. Moreover, we explained the two different aspects of the *what* class (here called declassification), one corresponds to fixing what can flow, and the other to fixing what should not flow, and we show that both can be modeled in abstract non-interference, while delimited release and relaxed non-interference can only fix what is *allowed* to flow. Finally, since abstract non-interference is in both the classes, we can study the relation between these two different ways of weakening non-interference, and indeed in [11] the authors proved that the derivation of *what* flows and the derivation of *who* can observe, is formalized in terms of a Galois connection between the two constructors introduced in [9]. As future work, in the same spirit of [11], we would like to understand the relation existing between declassification via allowing and declassification via blocking, in order, for example to derive the information to block from the maximal amount of information disclosed, and vice versa.

In Table 3 we summarize the comparisons made in this paper. In particular we use the notation $\equiv$ for saying that two notions characterize the same end-to-end security policy, and *Id* corresponds to the identity relation/closure.

Acknowledgments

I would like to thank Anindya Banerjee and Roberto Giacobazzi for their insightful comments, and the anonymous referees for their helpful suggestions for improvement.

References

1. D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE Corp. Badford, MA, 1973.
2. S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conf. on Computer and Communications Security*, pages 198–209. ACM-Press, NY, October 2004.
3. D. Clark, S. Hunt, and P. Malacaria. Quantified interference: Information theory and information flow (extended abstract). In *Workshop on Issues in the Theory of Security, WITS*, 2004.
4. E. S. Cohen. Information transmission in sequential programs. *Foundations of Secure Computation*, pages 297–335, 1978.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of Conf. Record of the 4th ACM Symp. on Principles of Programming Languages* (*POPL '77*), pages 238–252. ACM Press, New York, 1977.
6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of Conf. Record of the 6th ACM Symp. on Principles of Programming Languages* (*POPL '79*), pages 269–282. ACM Press, New York, 1979.
7. D. E. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
8. A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 1–17. IEEE Computer Society Press, 2002.
9. R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*, pages 186–197. ACM-Press, NY, 2004.
10. R. Giacobazzi and I. Mastroeni. Proving abstract non-interference. In *Annual Conference of the European Association for Computer Science Logic (CSL'04)*, volume 3210, pages 280–294. Springer-Verlag, 2004.
11. R. Giacobazzi and I. Mastroeni. Adjoining declassification and attack models by abstract interpretation. In *Proc. of the European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 295–310. Springer-Verlag, 2005.
12. R. Giacobazzi and I. Mastroeni. Timed abstract non-interference. In *Proc. of The International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'05)*, Lecture Notes in Computer Science. Springer-Verlag, 2005. To appear.
13. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.

14. S. Hunt and I. Mastroeni. The PER model of abstract non-interference. In *Proc. of The 12th Internat. Static Analysis Symp. (SAS'05)*, volume 3672 of *Lecture Notes in Computer Science*, pages 171–185. Springer-Verlag, 2005.
15. R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37:113–138, 2000.
16. P. Laud. Semantics and program analysis of computationally secure information flow. In *In Programming Languages and Systems, 10th European Symp. On Programming, ESOP*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag, 2001.
17. P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. of the 32st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, pages 158–170. ACM-Press, NY, 2005.
18. G. Lowe. Quantifying information flow. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 18–31. IEEE Computer Society Press, 2002.
19. A.C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. IEEE Symp. on Security and Privacy.* IEEE Computer Society Press, 2004.
20. F. Ranzato and F. Tapparo. Strong preservation as completeness in abstract interpretation. In D. Schmidt, editor, *Proc. of the 13th European Symposium on Programming (ESOP'04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 18–32. Springer-Verlag, 2004.
21. A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. of the International Symp. on Software Security (ISSS'03)*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer-Verlag, October 2004.
22. A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE J. on selected ares in communications*, 21(1):5–19, 2003.
23. A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.
24. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. of* 18^{th} *IEEE Computer Security Foundations Workshop (CSFW-18).* IEEE Comp. Soc. Press, 2005.
25. D. Volpano. Safety versus secrecy. In *Proc. of the 6th Static Analysis Symp. (SAS'99)*, volume 1694 of *Lecture Notes in Computer Science*, pages 303–311. Springer-Verlag, 1999.
26. S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 15–23. IEEE Computer Society Press, 2001.

Author Index