# Lecture Notes in Computer Science 4634

Hanne Riis Nielson   Gilberto Filé (Eds.)

# Static Analysis

14th International Symposium, SAS 2007
Kongens Lyngby, Denmark, August 22-24, 2007
Proceedings

Springer

Volume Editors

Hanne Riis Nielson
Technical University of Denmark, Informatics and Mathematical Modelling
Richard Petersens Plads, 2800 Kongens Lyngby, Denmark
E-mail: riis@imm.dtu.dk

Gilberto Filé
University of Padova, Department of Pure and Applied Mathematics
via Trieste 63, 35121 Padova, Italy
E-mail: gilberto@math.unipd.it

Library of Congress Control Number: 2007932181

CR Subject Classification (1998): D.3.2-3, F.3.1-2, I.2.2, F.4.2, D.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-74060-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-74060-5 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springer.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12103201 06/3180 5 4 3 2 1 0

# Preface

The aim of static analysis is to develop principles, techniques and tools for validating properties of programs, for designing semantics-based transformations of programs and for obtaining high-performance implementations of high-level programming languages. Over the years the series of static analysis symposia has served as the primary venue for presentation and discussion of theoretical, practical and innovative advances in the area.

This volume contains the papers accepted for presentation at the 14th International Static Analysis Symposium (SAS 2007). The meeting was held August, 22–24, 2007, at the Technical University of Denmark (DTU) in Kongens Lyngby, Denmark. In response to the call for papers, 85 submissions were received. Each submission was reviewed by at least 3 experts and, based on these reports, 26 papers were selected after a week of intense electronic discussion using the EasyChair conference system. In addition to these 26 papers, this volume also contains contributions by the two invited speakers: Frank Tip (IBM T. J. Watson Research Center, USA) and Alan Mycroft (Cambridge University, UK).

On the behalf of the Program Committee, the Program Chairs would like to thank all the authors who submitted their work to the conference and also all the external referees who have been indispensable for the selection process. Special thanks go to Terkel Tolstrup and Jörg Bauer, who helped in handing the submitted papers and in organizing the structure of this volume. We would also like to thank the members of the Organizing Committee at DTU for their great work. Finally we want to thank the PhD school ITMAN at DTU for financial support.

SAS 2007 was held concurrently with LOPSTR 2007, the International Symposium on Logic-Based Program Synthesis and Transformation.

June 2007

Hanne Riis Nielson<br>
Gilberto Filé

# Organization

## Program Chairs

| | |
|---|---|
| Gilberto Filé | University of Padova, Italy |
| Hanne Riis Nielson | Technical University of Denmark, Denmark |

## Program Committee

| | |
|---|---|
| Agostino Cortesi | University Ca'Foscari of Venice, Italy |
| Patrick Cousot | École Normale Supérieure, France |
| Manuel Fähndrich | Microsoft Research, USA |
| Roberto Giacobazzi | University of Verona, Italy |
| Chris Hankin | Imperial College, UK |
| Manuel Hermenegildo | Technical University of Madrid, Spain |
| Jens Knoop | Technical University of Vienna, Austria |
| Naoki Kobayashi | Tohoku University, Japan |
| Julia Lawall | Copenhagen University, Denmark |
| Andreas Podelski | University of Freiburg, Germany |
| Jakob Rehof | University of Dortmund, Germany |
| Radu Rugina | Cornell University, USA |
| Mooly Sagiv | Tel-Aviv University, Israel |
| David Schmidt | Kansas State University, USA |
| Helmut Seidl | Technical University of Munich, Germany |
| Harald Søndergaard | University of Melbourne, Australia |
| Kwangkeun Yi | Seoul National University, Korea |

## Steering Committee

| | |
|---|---|
| Patrick Cousot | École Normale Supérieure, France |
| Gilberto Filé | University of Padova, Italy |
| David Schmidt | Kansas State University, USA |

## Organizing Committee

Christian W. Probst
Sebastian Nanz
Flemming Nielson
Henrik Pilegaard
Terkel K. Tolstrup
Eva Bing
Elsebeth Strøm

## Referees

Luca de Alfaro
Rajeev Alur
Jesper Andersen
James Avery
Domagoj Babic
Roberto Bagnara
Josh Berdine
Julien Bertrane
Sapan Bhatia
Bruno Blanchet
Chiara Braghin
Bruno De Bus
Cristiano Calgano
Manuel Carro
Amadeo Casas
Bor-Yuh Evan Chang
Byron Cook
Silvia Crafa
Hyunjun Eo
M. Anton Ertl
Manuel Fahndrich
Xinyu Feng
Jerome Feret
Pietro Ferrara
Andrea Flexeder
Lars-Ake Fredlund
Samir Gemaim
Thomas Gawlitza
Andy Gordon
Alexey Gotsman
Andreas Griesmayer
Sumit Gulwani
Masami Hagiya
Hwansoo Han
Trevor Hansen
Rene Rydhof Hansen
Haifeng He
Patricia Hill
Yungbum Jung
Deepak Kapur
Deokhwan Kim
Andreas Krall
Shuvendu Lahiri
Akash Lal
Michel Leconte
Oukseh Lee
Heejong Lee
Tal Lev-Ami
Francesco Logozzo
Flaminia Luccio
Matteo Maffei
Rupak Majumdar
Roman Manevich
Mark Marron
Matthieu Martel
Florian Martin
Isabella Mastroeni
Laurent Mauborgne
Richard Mayr
Mario Mendez-Lojo
Alessio Merlo
Yasuhiko Minamide
Antoine Mine
David Monniaux
Sebastian Nanz
Jorge Navas
Nicholas Nethercote
Flemming Nielson
Hakjoo Oh
Daejun Park
Sungwoo Park
Michael Petter
Alessandra Di Pierro
Mila Dalla Preda
Franz Puntigam
Shaz Qadeer
Noam Rinetzky
Xavier Rival
Enric Rodriguez-Carbonell
Francesca Rossi
Andrey Rybalchenko
Oliver Rüthing
Cesar Sanchez
Peter Schachte
Markus Schordan
Stefan Schwoon
Roberto Segala
Sunae Seo
Elodie-Jane Sims
Zoltan Somogyi
Fausto Spoto
Manu Sridharan
Bjarne Steensgaard
Darko Stefanovic
Peter Stuckey
Kohei Suenaga
Yoshinori Tanabe
Francesco Tapparo
Makoto Tatsuta
Tachio Terauchi
Terkel Tolstrup
Tullio Vardanega
Kumar Neeraj Verma
Uwe Waldmann
Thomas Wies
Herbert Wiklicky
Eran Yahav
Greta Yorsh
Enea Zaffanella
Francesco Ranzato

# Table of Contents

## Invited Papers

Refactoring Using Type Constraints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
*Frank Tip*

Programming Language Design and Analysis Motivated by Hardware Evolution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
*Alan Mycroft*

## Contributed Papers

A Compilation Model for Aspect-Oriented Polymorphically Typed Functional Languages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
*Kung Chen, Shu-Chun Weng, Meng Wang, Siau-Cheng Khoo, and Chung-Hsin Chen*

Lattice Automata: A Representation for Languages on Infinite Alphabets, and Some Applications to Verification . . . . . . . . . . . . . . . . . . . 52
*Tristan Le Gall and Bertrand Jeannet*

Compositional Verification and 3-Valued Abstractions Join Forces . . . . . . 69
*Sharon Shoham and Orna Grumberg*

Formalised Inductive Reasoning in the Logic of Bunched Implications . . . 87
*James Brotherston*

Optimal Abstraction on Real-Valued Programs . . . . . . . . . . . . . . . . . . . . . . 104
*David Monniaux*

Taming the Wrapping of Integer Arithmetic . . . . . . . . . . . . . . . . . . . . . . . . 121
*Axel Simon and Andy King*

Under-Approximations of Computations in Real Numbers Based on Generalized Affine Arithmetic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137
*Eric Goubault and Sylvie Putot*

A Framework for End-to-End Verification and Evaluation of Register Allocators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153
*V. Krishna Nandivada, Fernando Magno Quintão Pereira, and Jens Palsberg*

A New Algorithm for Identifying Loops in Decompilation . . . . . . . . . . . . . 170
*Tao Wei, Jian Mao, Wei Zou, and Yu Chen*

Accelerated Data-Flow Analysis ........................................ 184
*Jérôme Leroux and Grégoire Sutre*

Abstract Error Projection ........................................ 200
*Akash Lal, Nicholas Kidd, Thomas Reps, and Tayssir Touili*

Precise Thread-Modular Verification ........................................ 218
*Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko*

Modular Safety Checking for Fine-Grained Concurrency .............. 233
*Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis*

Static Analysis of Dynamic Communication Systems by Partner Abstraction ........................................ 249
*Jörg Bauer and Reinhard Wilhelm*

Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis ........................................ 265
*Ben Hardekopf and Calvin Lin*

Hierarchical Pointer Analysis for Distributed Programs ............... 281
*Amir Kamil and Katherine Yelick*

Semantics-Based Transformation of Arithmetic Expressions ........... 298
*Matthieu Martel*

A Fast Implementation of the Octagon Abstract Domain on Graphics Hardware ........................................ 315
*Francesco Banterle and Roberto Giacobazzi*

Fixpoint-Guided Abstraction Refinements ........................................ 333
*Patrick Cousot, Pierre Ganty, and Jean-François Raskin*

Guided Static Analysis ........................................ 349
*Denis Gopan and Thomas Reps*

Program Analysis Using Symbolic Ranges ........................................ 366
*Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta*

Shape Analysis with Structural Invariant Checkers .................. 384
*Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula*

Footprint Analysis: A Shape Analysis That Discovers Preconditions .... 402
*Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang*

Arithmetic Strengthening for Shape Analysis ...................... 419
*Stephen Magill, Josh Berdine, Edmund Clarke, and Byron Cook*

Astrée: From Research to Industry ........................................ 437
*David Delmas and Jean Souyris*

Magic-Sets Transformation for the Analysis of Java Bytecode .......... 452
*Étienne Payet and Fausto Spoto*

**Author Index** .................................................... 469

# Refactoring Using Type Constraints*

Frank Tip

IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA
ftip@us.ibm.com

**Abstract.** Type constraints express subtype-relationships between the types of program expressions that are required for type-correctness, and were originally proposed as a convenient framework for solving type checking and type inference problems. In this paper, we show how type constraints can be used as the basis for practical refactoring tools. In our approach, a set of type constraints is derived from a type-correct program $P$. The main insight behind our work is the fact that $P$ constitutes just one solution to this constraint system, and that alternative solutions may exist that correspond to refactored versions of $P$. We show how a number of refactorings for manipulating types and class hierarchies can be expressed naturally using type constraints. Several refactorings in the standard distribution of Eclipse are based on our results.

## 1 Introduction

Refactoring is the process of applying behavior-preserving transformations (called "refactorings") to a program's source code with the objective of improving that program's design. Common reasons for refactoring include the elimination of undesirable program characteristics such as duplicated code, making existing program components reusable in new contexts, and breaking up monolithic systems into components. Pioneered in the early 1990s by Opdyke et al. [15,16] and by Griswold et al. [9,10], the field of refactoring received a major boost with the emergence of code-centric design methodologies such as extreme programming [2] that advocate continuous improvement of code quality. Fowler [7] and Kerievsky [12] authored popular books that classify many widely used refactorings, and Mens and Tourwé [14] presented a survey of the field.

Refactoring is usually presented as an interactive process where the programmer takes the initiative by indicating a point in the program where a specific transformation should be applied. Then, the programmer must verify if a number of specified preconditions hold, and, assuming this is the case, apply a number of prescribed editing steps. However, checking the preconditions may involve nontrivial analysis, and the number of editing steps may be significant. Therefore, automated tool support for refactoring is highly desirable, and has become a standard feature of modern development environments such as Eclipse (`www.eclipse.org`) and IntelliJ IDEA (`www.jetbrains.com/idea`).

* This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 1–17, 2007.

The main observation of this paper is that, for an important category of refactorings related to the manipulation of class hierarchies and types, the checking of preconditions and computation of required source code modifications can be expressed as a system of type constraints. Type constraints [17] are a formalism for expressing subtype-relationships between the types of program elements that must be satisfied in order for a program construct to be type-correct, and were originally proposed as a means for expressing type checking and type inference problems. In our work, a system of type constraints is derived from a program to reason about the correctness of refactorings. Specifically, we derive a set of type constraints from a program $P$ and observe that, while the types and class hierarchy of $P$ constitute one solution to the constraint system, alternative solutions may exist that correspond to refactored versions of $P$.

We show how several refactorings for manipulating class hierarchies and types can be expressed in terms of type constraints. This includes refactorings that: (i) introduce interfaces and supertypes, move members up and down in the class hierarchy, and change the declared type of variables, (ii) introduce generics, and (iii) replace deprecated classes with ones that are functionally equivalent. Several refactorings[1] in the Eclipse 3.2 distribution are based on the research presented in this paper. Our previous papers [22,3,8,1,13], presented these refactorings in detail, along with experimental evaluations. This paper presents an informal overview of the work and uses a running example to show how different refactorings require slight variations on the basic type constraints model.

## 2 Type Constraints

Type constraints are a formalism for expressing subtype relationships between the types of declarations and expressions, and were originally proposed as a means for stating type-checking and type inference problems [17]. In the basic model, a *type constraint* has of one of the following forms:

| | |
|---|---|
| $\alpha = \alpha'$ | type $\alpha$ must be the same as type $\alpha'$ |
| $\alpha < \alpha'$ | type $\alpha$ must be a proper subtype of type $\alpha'$ |
| $\alpha \leq \alpha'$ | type $\alpha$ must be the same as, or a subtype of type $\alpha'$ |
| $\alpha \leq \alpha_1$ **or** $\cdots$ **or** $\alpha \leq \alpha_k$ | $\alpha \leq \alpha_i$ must hold for at least one $i$, $(1 \leq i \leq k)$ |

Here, $\alpha$, $\alpha'$, ... are *constraint variables* that represent the types associated with program constructs. In this paper, $M$ denotes a method (with associated signature and type information), $F$ denotes a field, $C$ denotes a class, $I$ denotes an interface, $T$ denotes a class or an interface, and $E$ denotes an expression. Constraint variables are of one of the following forms:

| | | | |
|---|---|---|---|
| $T$ | a type constant | $[F]$ | the declared type of field $F$ |
| $[E]$ | the type of an expression $E$ | $Decl(M)$ | the type in which method $M$ is declared |
| $[M]$ | the declared return type of method $M$ | $Decl(F)$ | the type in which field $F$ is declared |

[1] This includes the Extract Interface, Generalize Declared Type, and Infer Generic Type Arguments refactorings presented in this paper, among others.

| **program construct** | **implied type constraint(s)** | |
|---|---|---|
| assignment $E_1 = E_2$ | $[E_2]\leq[E_1]$ | (1) |
| method call $E.m(E_1, \cdots, E_n)$ | $[E.m(E_1, \cdots, E_n)]=[M]$ | (2) |
| to a virtual method $M$ | $[E_i]\leq[\mathit{Param}(M, i)]$ | (3) |
| where $\mathit{RootDefs}(M) = \{ M_1, \cdots, M_k \}$ | $[E]\leq \mathit{Decl}(M_1)$ **or** $\cdots$ **or** $[E]\leq \mathit{Decl}(M_k)$ | (4) |
| access $E.f$ to field $F$ | $[E.f]=[F]$ | (5) |
| | $[E]\leq \mathit{Decl}(F)$ | (6) |
| `return` $E$ in method $M$ | $[E]\leq[M]$ | (7) |
| $M'$ overrides $M$, | $[\mathit{Param}(M', i)]=[\mathit{Param}(M, i)]$ | (8) |
| $M' \neq M$ | $[M']\leq[M]$ | (9) |
| $F'$ hides $F$ | $\mathit{Decl}(F')<\mathit{Decl}(F)$ | (10) |
| constructor call `new` $C(E_1, \cdots, E_n)$ | [`new` $C(E_1, \cdots, E_n)]=C$ | (11) |
| to constructor $M$ | $[E_i]\leq[\mathit{Param}(M, i)]$ | (12) |
| direct call | $[E.m(E_1, \cdots, E_n)]=[M]$ | (13) |
| $E.m(E_1, \cdots, E_n)$ | $[E_i]\leq[\mathit{Param}(M, i)]$ | (14) |
| to method $M$ | $[E]\leq \mathit{Decl}(M)$ | (15) |
| implicit declaration of `this` in method $M$ | [`this`]$=\mathit{Decl}(M)$ | (16) |

**Fig. 1.** Type constraints for a set of core Java language features

Type constraints are generated from a program's abstract syntax tree in a syntax-directed manner, and encode relationships between the types of declarations and expressions that must be satisfied in order to preserve type correctness or program behavior. Figure 1 shows rules that generate constraints from a representative set of program constructs.

For example, rule (1) states that, for an assignment $E_1 = E_2$, a constraint $[E_2]\leq[E_1]$ is generated. Intuitively, this captures the requirement that the type of the right-hand side $E_2$ be a subtype of the type of the left-hand side $E_1$ because otherwise the assignment would not be type correct. In the rules discussed below, $\mathit{Param}(M, i)$ denotes the $i$-th formal parameter of method $M$. For a call $E.m(\cdots)$ to a virtual method $M$, we have that: the type of the call-expression is the same as $M$'s return type (rule (2)[2]), the type of each actual parameter must be the same as, or a subtype of the corresponding formal parameter (rule (3)), and a method with the same signature as $M$ must be declared in $[E]$ or one of its supertypes (rule (4)). Rule (4) determines a set of methods $M_1, \cdots, M_k$ overridden by $M$ using Definition 1 below, and requires $[E]$ to be a subtype of one or more[3] of $\mathit{Decl}(M_1), \cdots, \mathit{Decl}(M_k)$. In this definition, a virtual method $M$ in type $C$ overrides a virtual method $M'$ in type $B$ if $M$ and $M'$ have identical signatures and $C$ is equal to $B$ or $C$ is a subtype of $B$.

**Definition 1 (RootDefs).** *Let $M$ be a method. Define:*
$\mathit{RootDefs}(M) = \{ M' | \, M$ *overrides* $M'$*, and there exists no*
$M''$ $(M'' \neq M')$ *such that* $M'$ *overrides* $M'' \}$

[2] Rules (2), (5), (13), (11), and (16) *define* the type of certain kinds of expressions. While not very interesting by themselves, these rules are essential for defining the relationships between the types of expressions and declaration elements.

[3] In cases where a referenced method does not occur in a supertype of $[E]$, the *RootDefs*-set defined in Definition 1 will be empty, and an **or**-constraint with zero branches will be generated. Such constraints are never satisfied and do not occur in our setting because we assume the original program to be type-correct.

Changing a parameter's type need not affect type-correctness, but may affect virtual dispatch (and program) behavior. Hence, we require that types of corresponding parameters of overriding methods be identical (rule (8)). As of Java 5.0, return types in overriding methods may be covariant (rule (9)). Rule (16) defines the type of a `this` expression to be the class that declares the associated method. The constraint rules for several features (e.g., casts) have been omitted due to space limitations and can be found in our earlier papers.

## 3 Refactorings for Generalization

Figure 2 shows a Java program that was designed to illustrate the issues posed by several different refactorings. The program declares a class `Stack` representing a stack, with methods `push()`, `pop()`, and `isEmpty()` with the expected behaviors, methods `moveFrom()` and `moveTo()` for moving an element from one stack to another, and a static method `print()` for printing a stack's contents. Also shown is a class `Client` that creates a stack, pushes the integer `1` onto it, then creates another stack onto which it pushes the values `2.2` and `3.3`. The elements of the second stack are then moved to the first, the contents of one of the stacks is printed, and the elements of the first stack are transferred into a `Vector` whose contents are displayed in a tree. Executing the program creates a graphical representation of a tree containing, from top to bottom, nodes 2.2, 3.3, and 1.

### 3.1 Extract Interface

One possible criticism about the code in Figure 2 is the fact that class `Client` explicitly refers to class `Stack`. Such explicit dependences on concrete data structures are generally frowned upon because they make code less flexible. The Extract Interface refactoring aims to address this issue by introducing an interface that declares a subset of the methods in a class, and updating references in client code to refer to the interface instead of the class wherever possible. Let us assume that the programmer has decided that it would be desirable to create an interface `IStack` that declares all of `Stack`'s instance methods, and to update references to `Stack` to refer to `IStack` instead, as shown in Figure 3 (code fragments changed by the application of Extract Interface are underlined). Observe that `s1`, `s3`, and `s4` are the only variables for which the type has been changed to `IStack`. Changing the type of `s2` or `s5` to `IStack` would result in type errors. In particular, changing `s5`'s type to `IStack` results in an error because field `v2`, which is not declared in `IStack`, is accessed from `s5` on line 45.

Using type constraints, it is straightforward to compute the declarations that can be updated to refer to `IStack` instead of `Stack`. Figure 4(a) shows some of the type constraints generated for declarations and expressions of type `Stack` in the program of Figure 2, according to the the rules of Figure 1. It is important to note that the constraints were generated *after* adding interface `IStack` to the class hierarchy. Now, from the constraints of Figure 4(a), it is easy to see that `Stack`≤[`s2`]≤[`s5`]≤`Stack` and hence that the types of `s2` and `s5` have to remain

```
[1]  class Client {
[2]    public static void main(String[] args){
[3]      Stack s1 = new Stack();
[4]      s1.push(new Integer(1));
[5]      Stack s2 = new Stack();
[6]      s2.push(new Float(2.2));
[7]      s2.push(new Float(3.3));
[8]      s1.moveFrom(s2);
[9]      s2.moveTo(s1);
[10]     Stack.print(s2);
[11]     Vector v1 = new Vector(); /* A1 */
[12]     while (!s1.isEmpty()){
[13]       Number n = (Number)s1.pop();
[14]       v1.add(n);
[15]     }
[16]     JFrame frame = new JFrame();
[17]     frame.setTitle("Example");
[18]     frame.setSize(300, 100);
[19]     JTree tree = new JTree(v1);
[20]     frame.add(tree, BorderLayout.CENTER);
[21]     frame.setVisible(true);
[22]   }
[23] }
```

```
[24] class Stack {
[25]   private Vector v2;
[26]   public Stack(){
[27]     v2 = new Vector(); /* A2 */
[28]   }
[29]   public void push(Object o){
[30]     v2.addElement(o);
[31]   }
[32]   public Object pop(){
[33]     return v2.remove(v2.size()-1);
[34]   }
[35]   public void moveFrom(Stack s3){
[36]     this.push(s3.pop());
[37]   }
[38]   public void moveTo(Stack s4){
[39]     s4.push(this.pop());
[40]   }
[41]   public boolean isEmpty(){
[42]     return v2.isEmpty();
[43]   }
[44]   public static void print(Stack s5){
[45]     Enumeration e = s5.v2.elements();
[46]     while (e.hasMoreElements())
[47]       System.out.println(e.nextElement());
[48]   }
[49] }
```

**Fig. 2.** An example program. The allocation sites for the two `Vector` objects created by this program have been labeled `A1` and `A2` to ease the discussion of the Replace Class refactoring in Section 5.

```
class Client {
  public static void main(String[] args){
    IStack s1 = new Stack();
    s1.push(new Integer(1));
    Stack s2 = new Stack();
    s2.push(new Float(2.2));
    s2.push(new Float(3.3));
    s1.moveFrom(s2);
    s2.moveTo(s1);
    Stack.print(s2);
    Vector v1 = new Vector();
    while (!s1.isEmpty()){
      Number n = (Number)s1.pop();
      v1.add(n);
    }
    JFrame frame = new JFrame();
    frame.setTitle("Example");
    frame.setSize(300, 100);
    Component tree = new JTree(v1);
    frame.add(tree, BorderLayout.CENTER);
    frame.setVisible(true);
  }
}
interface IStack {
  public void push(Object o);
  public Object pop();
  public void moveFrom(IStack s3);
  public void moveTo(IStack s4);
  public boolean isEmpty();
}
```

```
class Stack implements IStack {
   private Vector v2;
   public Stack(){
     v2 = new Vector();
   }
   public void push(Object o){
     v2.addElement(o);
   }
   public Object pop(){
     return v2.remove(v2.size()-1);
   }
   public void moveFrom(IStack s3){
     this.push(s3.pop());
   }
   public void moveTo(IStack s4){
     s4.push(this.pop());
   }
   public boolean isEmpty(){
     return v2.isEmpty();
   }
   public static void print(Stack s5){
     Enumeration e = s5.v2.elements();
     while (e.hasMoreElements())
       System.out.println(e.nextElement());
   }
}
```

**Fig. 3.** The example program of Figure 2 after applying Extract Interface to class `Stack` (code fragments affected by this step are underlined), and applying Generalize Declared Type to variable `tree` (the affected code fragment is shown boxed)

| line(s) | constraint(s) | rule(s) |
|---|---|---|
| 3 | Stack≤[ s1 ] | (11),(1) |
| 4, 8, 12, 13 | [ s1 ]≤IStack | (4) |
| 5 | Stack≤[ s2 ] | (11),(1) |
| 6, 7, 9 | [ s2 ]≤IStack | (4) |
| 8,35 | [ s2 ]≤[ s3 ] | (3) |
| 9,38 | [ s1 ]≤[ s4 ] | (3) |
| 10,44 | [ s2 ]≤[ s5 ] | (14) |
| 36 | [ s3 ]≤IStack | (4) |
| 39 | [ s4 ]≤IStack | (4) |
| 45 | [ s5 ]≤Stack | (6) |

**(a)**

| line(s) | constraint(s) | rule applied |
|---|---|---|
| 19 | JTree≤[ tree ] | (11),(1) |
| 20 | [ tree ]≤Component | (12) |
| 11 | Vector≤[ v1 ] | (11),(1) |
| 14 | [ v1 ]≤Collection | (4) |
| 19 | [ v1 ]≤Vector | (12) |
| 27 | Vector≤[ v2 ] | (11),(1) |
| 30 | [ v2 ]≤Vector | (4) |
| 33, 42 | [ v2 ]≤Collection | (4) |

**(b)**

**Fig. 4.** (a) Type constraints generated for the application of the Extract Interface refactoring to the program of Figure 2 (only nontrivial constraints related to variables s1–s5 are shown). (b) Type constraints used for the application of Generalize Declared Type (only nontrivial constraints related to variables tree, v1, and v2 are shown). Line numbers refer to Figure 2, and rule numbers to rules of Figure 1.

Stack. However, the types of s1 and s4 are less constrained ([s1]≤[s4]≤IStack) implying that type IStack may be used for these variables. In general, the types of variables may not be changed independently. For example, changing s1's type to IStack but leaving s4's type unchanged results in a type-incorrect program. In a previous paper [22], we presented an algorithm for computing the maximal set of variables whose type can be updated to refer to a newly extracted interface.

### 3.2 Generalize Declared Type

Another possible criticism of the program of Figure 2 is the fact that the types of some variable declarations in the program of Figure 2 are overly specific. This is considered undesirable because it reduces flexibility. The Generalize Declared Type refactoring in Eclipse lets a programmer select a declaration, and determines whether its type can be generalized without introducing type errors or behavioral changes. If so, the programmer may choose from the alternative permissible types. Using this refactoring, the type of variable tree can be updated to refer to Component instead of JTree without affecting type-correctness or program behavior, as is indicated by a box in Figure 3. This, in turn, would enable one to vary the implementation to use, say, a JList instead of a JTree in Client.main(). In some situations, the type of a variable cannot be generalized. For example, changing the type of v2 to Collection (or to any other supertype of Vector) would result in a type error because the method addElement(), which is not declared in any supertype of Vector, is invoked on v2 on line 30. Furthermore, the type of v1 cannot be generalized because, on line 19, v1 is passed as an argument to the constructor JTree(Vector). JTree is part of the standard Java libraries (for which we cannot change the source code), and the fact that its constructor expects a Vector implies that a more general type cannot be used.

Figure 4(b) shows the constraints generated from the example program of Figure 2 for variables tree, v1, and v2. Note that, for parameters of methods in external classes such as the constructor of JTree, we must include constraints that constrain these parameters to have their originally declared type, because the source code in class libraries cannot be changed. Therefore, we have that: JTree$\leq$[tree]$\leq$Component, Vector$\leq$[v1]$\leq$Vector, and Vector$\leq$[v2]$\leq$Vector. In other words, the types of v1 and v2 must be exactly Vector, but for tree we may choose any supertype of JTree that is a subtype of Component.

### 3.3 Other Refactorings for Generalization

Several other refactorings related to generalization can be modeled similarly. For example, the Pull Up Members refactoring is concerned with moving methods and fields from a class to one of its superclasses. For this refactoring, we leave the types of variables constant by including constraints that require variables to have their originally declared type while allowing the locations of methods and fields to vary by leaving constraint variables of the form *Decl*(.) unconstrained.

## 4 Refactorings That Introduce Generics

Generics were introduced in Java 5.0 to enable the creation of reusable class libraries with compiler-enforced type-safe usage. For example, an application that instantiates Vector<E> with, say, String, obtaining Vector<String>, can only add and retrieve Strings. In the previous, non-generic version of this class, the signatures of access methods such as Vector.get() refer to type Object, which prevents the compiler from ensuring the type-safety of vector operations, and therefore down-casts to String are needed to recover the type of retrieved elements. When a programmer makes a mistake, such downcasts fail at runtime, with ClassCastExceptions.

Donovan et al. [5] identified two refactoring problems related to the introduction of generics. The *parameterization problem* consists of adding type parameters to an existing class definition so that it can be used in different contexts without the loss of type information. Once a class has been parameterized, the *instantiation problem* is the task of determining the type arguments that should be given to instances of the generic class in client code. The former problem subsumes the latter because the introduction of type parameters often requires the instantiation of generic classes.

The Introduce Type Parameter refactoring developed recently by Kieżun et al. [13] provides a solution to the parameterization problem in which the programmer selects a declaration for which the type is to be replaced with a new formal type parameter. As we shall see shortly, this may involve nontrivial changes to other declarations (e.g., by introducing wildcard types [24]). Fuhrer et al. [8] proposed a solution to the instantiation problem that forms the basis for the Infer Generic Type Arguments refactoring in Eclipse.

The right column of Figure 5 shows class Stack after applying Introduce Type Parameter to the formal parameter of method Stack.push() (for the purposes of this example, it is assumed that class Stack is analyzed in isolation). Underlining is used to indicate changes w.r.t. the version of Stack in Figure 2.

As can be seen in the figure, a new type parameter T1 was added to class Stack, and T1 is used as the type for the parameter of Stack.push(), for the return type of Stack.pop(), and for the type of field v2. A more interesting change can be seen in the moveFrom(), moveTo(), and print() methods. Here, the parameters now have wildcard types Stack<? extends T1>, Stack<? super T1>, and Stack<?>, respectively. As we shall see shortly, this allows for greater flexibility when refactoring class Client because it enables the transfer of elements between the two stacks without the loss of precision in their declared types.

The left column of Figure 5 shows the result of applying Infer Generic Type Arguments to the example program after the parameterization of Stack. Observe that the types of s1 and s2 are now Stack<Number> and Stack<Float>, and that the downcast on line 13 that was present originally has been removed. This result was enabled directly by the introduction of wildcard types in Stack.moveFrom() and Stack.moveTo(). If the formal parameters of these methods had been changed to Stack<T1> instead, Java's typing rules would have required Vector<Number> for the types of s1 and s2, making it impossible to remove the downcast.

```
class Client {
  public static void main(String[] args){
    Stack<Number> s1 = new Stack<Number>();
    s1.push(new Integer(1));
    Stack<Float> s2 = new Stack<Float>();
    s2.push(new Float(2.2));
    s2.push(new Float(3.3));
    s1.moveFrom(s2);
    s2.moveTo(s1);
    Stack.print(s2);
    Vector<Number> v1 = new Vector<Number>();
    while (!s1.isEmpty()){
      Number n = s1.pop();
      v1.add(n);
    }
    JFrame frame = new JFrame();
    frame.setTitle("Example");
    frame.setSize(300, 100);
    JTree tree = new JTree(v1);
    frame.add(tree, BorderLayout.CENTER);
    frame.setVisible(true);
  }
}
```

```
class Stack<T1> {
   private Vector<T1> v2;
   public Stack(){
     v2 = new Vector<T1>();
   }
   public void push(T1 o){
     v2.addElement(o);
   }
   public T1 pop(){
     return v2.remove(v2.size()-1);
   }
   public void moveFrom(Stack<? extends T1> s3){
     this.push(s3.pop());
   }
   public void moveTo(Stack<? super T1> s4){
     s4.push(this.pop());
   }
   public boolean isEmpty(){
     return v2.isEmpty();
   }
   public static void print(Stack<?> s5){
     Enumeration<?> e = s5.v2.elements();
     while (e.hasMoreElements())
       System.out.println(e.nextElement());
   }
}
```

**Fig. 5.** The example program after the application of Introduce Type Parameter to the formal parameter of Stack.push(), followed by an application of Infer Generic Type Arguments to the entire application

### 4.1 Infer Generic Type Arguments

The Infer Generic Type Arguments refactoring requires a minor extension of the type constraint formalism of Section 2, which we illustrate by way of our running example. Some technical details are not discussed due to space limitations, and can be found in a previous paper [8].

In order to reason about type parameters, we introduce a new kind of constraint variable. These constraint variables are of the form $T(x)$, representing the type that is bound to formal type parameter $T$ in the type of $x$. For example, if we have a parameterized class `Vector<E>` and a variable $v$ of type `Vector<String>`, then $\texttt{E}(v) = \texttt{String}$. We also need additional rules for generating type constraints to ensure that the appropriate values are inferred for the new constraint variables. We now give a few examples to illustrate how these rules are *inferred* from method signatures in parameterized classes. In giving these examples, we assume that class `Stack` has already been parameterized as in the right column of Figure 5 (either manually, or using the Introduce Type Parameter refactoring presented in Section 4.2).

*Example 1.* Consider the method call `s1.push(new Integer(1))` on line 4 in Figure 2. This call refers to the method `void Stack<T1>.push(T1 o)`. If `s1` is of a parameterized type, say, `Stack<`$\alpha$`>`, then this call can only be type-correct if $\texttt{Integer} \leq \alpha$ and this constraint is generated from rule (17) in Figure 6(a).

*Example 2.* Similarly, the call `s1.pop()` on line 13 refers to method `void Stack<T1>.pop()`. If `s1` is of some parametric type, say `Stack<`$\alpha$`>`, then $[\texttt{s1.pop()}] = \alpha$ and this constraint can be generated from rule (18).

*Example 3.* Consider the call `s1.moveFrom(s2)` on line 8. If we assume that `s1` and `s2` are of parameterized types `Stack<`$\alpha_1$`>` and `Stack<`$\alpha_2$`>`, for some $\alpha_1$, $\alpha_2$, then the call is type correct if we have that $\alpha_2 \leq \alpha_1$ and this constraint is generated from rule (19).

As can be seen from Figure 6(a), the rules for generating constraints have a regular structure, in which occurrences of type parameters in method signatures give rise to different forms of constraints. In the examples we have seen, type parameters occur as types of formal parameters, as return types, and as actual type parameters in the type of a formal parameter. Several other forms exist [8].

Figure 6(b) shows the constraints generated for the example. From these constraints, it follows that: $\texttt{Integer} \leq \texttt{T1(s1)}$, $\texttt{Float} \leq \texttt{T1(s2)}$, and $\texttt{T1(s2)} \leq \texttt{T1(s1)}$, and hence that $\texttt{Float} \leq \texttt{T1(s1)}$. Since `Number` is a common supertype of `Integer` and `Float`, a possible solution to this constraint system is:

$$\texttt{T1(s1)} \leftarrow \texttt{Number}, \texttt{T1(s2)} \leftarrow \texttt{Float}$$

However, several other solutions exist, such as the following uninteresting one:

$$\texttt{T1(s1)} \leftarrow \texttt{Object}, \texttt{T1(s2)} \leftarrow \texttt{Object}$$

Our current constraint solver relies on heuristics to guide it towards preferred solutions. The most significant of these heuristics are preferring more specific types over less specific ones, and avoiding marker interfaces such as `Serializable`.

Generating the refactored source code is now straightforward. The type of variable `s1` in the example program, for which we inferred $\texttt{T1(s1)} = \texttt{Number}$, is rewritten to `Stack<Number>`. Similarly, the types of `s2` and `v1` are rewritten to `Stack<Float>` and `Vector<Number>`, respectively. Furthermore, all downcasts are removed for which the type of the expression being cast is a subtype of the

| program construct | constraint(s) |
|---|---|
| method call $E_1$.`push`($E_2$) to<br>`void Stack<T1>.push(T1)` | $[E_2] \leq \texttt{T1}(E_1)$ (17) |
| method call $E$.`pop`() to<br>`void Stack<T1>.pop()` | $[E.\texttt{pop}()] = \texttt{T1}(E)$ (18) |
| method call $E_1$.`moveFrom`($E_2$) to<br>`void Stack<T1>.`<br>`moveFrom(Stack<? extends T1>)` | $\texttt{T1}(E_2) \leq \texttt{T1}(E_1)$ (19) |
| method call $E_1$.`moveTo`($E_2$) to<br>`void Stack<T1>.`<br>`moveTo(Stack<? super T1>)` | $\texttt{T1}(E_1) \leq \texttt{T1}(E_2)$ (20) |
| method call $E_1$.`add`($E_2$) to<br>`boolean Vector<E>.add(E)` | $[E_2] \leq \texttt{E}(E_1)$ (21) |

**(a)**

| line(s) | constraint(s) | rule(s) |
|---|---|---|
| 4 | `Integer`≤`T1(s1)` | (11),(17) |
| 6,7 | `Float`≤`T1(s2)` | (11),(17) |
| 8 | `T1(s2)`≤`T1(s1)` | (19) |
| 9 | `T1(s2)`≤`T1(s1)` | (20) |
| 13 | [ `s1.pop()` ]`=T(s1)` | (18) |
| 13 | `Number`≤`E(v1)` | (21) |

**(b)**

**Fig. 6.** (a) Additional constraint generation rules needed for the Infer Generic Type Arguments refactoring, automatically derived from method signatures (only constraints for methods used in the example program are shown). (b) Type constraints generated for the example program using the rules of (a). Only nontrivial constraints relevant to the inference of type parameters in uses of `Stack` and `Vector` are shown. Line numbers refer to Figure 2, and rule numbers refer to Figures 6(a) and 1.

target type. For example, for the downcast `(Number)s1.pop()` on line 13, we inferred $[\texttt{s1.pop}()] = \texttt{Number}$ enabling us to remove the cast.

### 4.2 Introduce Type Parameter

Consider a scenario where a programmer wants to apply Introduce Type Parameter to replace the type of the formal parameter `o` of `Stack.push()` with a new type parameter. Our solution requires a new form of constraint variable called context variable[4]. A *context variable* is of the form $\mathcal{I}_{\alpha'}(\alpha)$ and represents the interpretation of a constraint variable $\alpha$ in a *context* given by a constraint variable $\alpha'$. As an example, consider the type `Stack<T1>`. In the context of an instance `Stack<Number>`, the interpretation of `T1` is `Number`. Now, if we have a variable `x` of type `Stack<Number>`, then the interpretation of `T1` in the context of the type of `x` is `Number` and we will denote this fact by $\mathcal{I}_{[\texttt{x}]}(T1) = \texttt{Number}$. Here, $\mathcal{I}_{[\texttt{x}]}$ is an *interpretation function* that maps the formal type parameter[5] `T1` of `Stack` to the type with which it is instantiated in type [`x`]. For a more interesting example, consider the call `s1.push(new Integer(1))` on line 4 of Figure 2. For this call to be type-correct, the type `Integer` of actual parameter `new Integer(1)` must be a subtype of the formal parameter `o` of `Stack.push()` *in the context of the type of* `s1`. This can be expressed by a constraint $\texttt{Integer} \leq \mathcal{I}_{[\texttt{s1}]}([\texttt{o}])$. Note that we cannot simply require that $\texttt{Integer} \leq [\texttt{o}]$ because if `Stack` becomes a parameterized class `Stack<T1>`, and the type of `o` becomes `T1`, then `T1` is out of scope on line 4 (in addition, `Integer` is not a subtype of `T1`).

[4] Also required are *wildcard variables* to model cases where Java's typing rules *require* the introduction of wildcard types due to method overriding [13].

[5] For parameterized types with multiple type parameters such as `HashMap`, the interpretation function provides a binding for each of them [13].

| line(s) | constraint(s) | |
|---|---|---|
| 30 | $[\ \mathtt{o}\ ] \leq \mathtt{E(v2)}$ | (i) |
| 33 | $\mathtt{E(v2)} \leq [\ \mathtt{Stack.pop()}\ ]$ | (ii) |
| 36 | $\mathcal{I}_{[\mathtt{s3}]}([\mathtt{Stack.pop()}]) \leq [\ \mathtt{o}\ ]$ | (iii) |
| 39 | $[\ \mathtt{Stack.pop()}\ ] \leq \mathcal{I}_{[\mathtt{s4}]}([\mathtt{o}])$ | (iv) |

**(a)**

| constraint variable | inferred type |
|---|---|
| `o` | `T1` |
| `E(v2)` | `T1` |
| $[\ \mathtt{Stack.pop()}\ ]$ | `T1` |
| $\mathcal{I}_{[\mathtt{s3}]}([\mathtt{Stack.pop()}])$ | `? extends T1` |
| $\mathcal{I}_{[\mathtt{s4}]}([\mathtt{o}])$ | `? super T1` |

**(b)**

**Fig. 7.** (a) Type constraints generated for class `Stack` of Figure 2 when applying Introduce Type Parameter. (b) Solution to the constraints computed by our algorithm.

Figure 7(a) shows some of the constraints generated for class `Stack` of Figure 2. For these constraints, the algorithm by Kieżun et al. [13] computes the solution shown in Figure 7(b). This solution can be understood as follows. The type of `o` has become a new type parameter `T1` because this declaration was selected by the user. From constraints (i) and (ii) in Figure 7, it follows that `E(v2)` and $[\ \mathtt{Stack.pop()}\ ]$ must each be a supertype of `T1`, and from constraint (iii) it can be seen that $\mathcal{I}_{[\mathtt{s3}]}([\mathtt{Stack.pop()}])$ must be a subtype of `T1`. The only possible choices for $[\ \mathtt{Stack.pop()}\ ]$ are `T1` and `Object` because wildcard types are not permitted in this position, and `T1` is selected because the choice of `Object` would lead to a violation of constraint (iii).

Taking into account constraint (ii), it follows that $\mathtt{E(v2)} = \mathtt{T1}$. Now, for $\mathcal{I}_{\mathtt{s3}}([\mathtt{Stack.pop()}])$, the algorithm may choose any subtype of `T1`, and it heuristically[6] chooses `? extends T1`. Likewise, the type `? super T1` is selected for $\mathcal{I}_{[\mathtt{s4}]}([\mathtt{o}])$.

At this point, determining how the rewrite the source code is straightforward. From Figure 7(b), it can be seen that type of `o` and the return type of `Stack.pop()` become `T1`. Moreover, from $\mathtt{E(v2)} = \mathtt{T1}$, it follows that `v2` becomes `Vector<T1>`. The type of `s3` is rewritten to `Stack<? extends T1>` because the return type of `Stack.pop()` is `T1` and the type `? extends T1` was inferred for $\mathcal{I}_{[\mathtt{s3}]}([\mathtt{Stack.pop()}])$. By a similar argument, the type of `s4` is rewritten to `Stack<? super T1>`. The right column of Figure 5 shows the result.

## 5 A Refactoring for Replacing Classes

As applications evolve, classes are occasionally deprecated in favor of others with roughly the same functionality. In Java's standard libraries, for example, class `Hashtable` has been superseded by `HashMap`, and `Iterator` is now preferred over `Enumeration`. In such cases it is often desirable to migrate client applications to make use of the new idioms, but manually making the required changes can be labor-intensive and error-prone. In what follows, we will use the term *migration* to refer to the process of replacing the references to a *source class* with references to a *target class*.

[6] Other possible choices include `T1`, or a new type parameter that is a subtype of `T1`. The paper by Kieżun et al. [13] presents more details on the use of heuristics.

In the program of Figure 2, `Vector`s are used in two places (variable `v1` declared on line 11 and field `v2` declared on line 25). Class `ArrayList` was introduced in the standard libraries to replace `Vector`, and is considered preferable because its interface is minimal and matches the functionality of the `List` interface. `ArrayList` also provides unsynchronized access to a list's elements whereas all of `Vector`'s methods are `synchronized`, which results in unnecessary overhead when `Vector`s are used by only one thread. The example program illustrates several factors that complicate the migration from `Vector` to `ArrayList`:

- Some methods in `Vector` are not supported by `ArrayList`. E.g, the example program calls `Vector.addElement()` on line 30, a method not declared in `ArrayList`. In this case, the call can be replaced with a call to `ArrayList.add()`, but other cases require the introduction of more complex expressions, or preclude migration altogether.
- Opportunities for migration may be limited when applications interact with libraries. For example, variable `v1` declared on line 11 serves as the actual parameter in a call to a constructor `JTree(Vector)` on line 19. Changing the type of `v1` to any supertype of `Vector` would render this call type-incorrect. Hence, the allocation site labeled `A1` cannot be migrated to `ArrayList`.
- Migrating one class may require migrating another. Consider the call on line 45 to `Vector.elements()`, which returns an `Enumeration`. `ArrayList` does not declare this method, but its method `iterator()` returns an `Iterator`, an interface with similar functionality[7]. In this case, we can replace the call to `elements()` with a call to `iterator()`, *provided that* we replace the calls to `Enumeration.hasMoreElements()` and `Enumeration.nextElement()` on lines 46 and 47 with calls to `Iterator.hasNext()` and `Iterator.next()`.
- If a `Vector` is accessed concurrently, then preservation of synchronization behavior is important. This is accomplished by introducing *synchronization wrappers*. This issue does not arise in the program of Figure 2 because it is single-threaded; the paper by Balaban et al. [1] presents an example.

We have developed a Replace Class refactoring that addresses these migration problems. This refactoring relies on a *migration specification* that specifies for each method in the source class how it is to be rewritten. Figure 8 shows the fragments of the specification for performing the migration from `Vector` to `ArrayList` and from `Enumeration` to `Iterator` needed for the example program (the complete specification can be found in [1]). Migration specifications only have to be written *once* for each pair of (source,target) classes.

We adapt the type constraints formalism of Section 2 as follows to implement Replace Class. For each source class $S$ and target class $T$ in a migration, the type system is extended with types $S^\top$ and $S_\perp$, such that $S \leq S^\top$, $T \leq S^\top$, $S_\perp \leq S$,

[7] The methods `hasNext()` and `next()` in `Iterator` correspond to `hasMoreElements()` and `nextElement()` in `Enumeration`, respectively. `Iterator` declares an additional method `remove()` for the removal of elements from the collection being iterated over.

```
(1)  new Vector(), unsynchronized                      → new ArrayList()
(2)  new Vector(), synchronized                        → Collections.synchronizedList(
                                                             new ArrayList())
(3)  boolean Vector:receiver.add(Object:v)             → boolean receiver.add(v)
(4)  void Vector:receiver.addElement(Object:v)         → boolean receiver.add(v)
(5)  Object Vector:receiver.remove(int:i)              → Object receiver.remove(i)
(6)  int Vector:receiver.size()                        → int receiver.size()
(7)  boolean Vector:receiver.isEmpty()                 → boolean receiver.isEmpty()
(8)  Enumeration Vector:receiver.elements()            → Iterator receiver.iterator()
(9)  boolean Enumeration:receiver.hasMoreElements()    → boolean receiver.hasNext()
(10) Object Enumeration:receiver.nextElement()         → Object receiver.next()
```

**Fig. 8.** Specification used for migrating the example program

| **line(s)** | **constraint(s)** | |
|---|---|---|
| 11 | [ A1 ]$\leq$[ v1 ], [ A1 ]$\leq$Vector$^{\top}$, Vector$_{\perp}\leq$[ A1 ] | (i),(ii),(iii) |
| 19 | [ v1 ]$\leq$Vector | (iv) |
| 27 | [ A2 ]$\leq$[ v2 ], [ A2 ]$\leq$Vector$^{\top}$, Vector$_{\perp}\leq$[ A2 ] | (v),(vi),(vii) |
| 30 | [ o ]$\leq$Object | (viii) |
| 33,42 | [ v2 ]$\leq$Collection | (ix) |
| 45 | [ s5.v2 ]=Vector → [ s5.v2.elements() ]=Enumeration | (x) |
| 45 | [ s5.v2 ]=ArrayList → [ s5.v2.elements() ]=Iterator | (xi) |

**Fig. 9.** Some of the type constraints generated for the application of the Replace Class refactoring to the program of Figure 2

$S_{\perp}\leq T$[8]. Moreover, rule (11) of Figure 1 is adapted to generate constraints for allocation sites that permit the migration from source types to target types. For example, constraints (ii) and (iii) in Figure 9 are generated for the allocation site labeled `A1` on line 11 in Figure 2.

For a migration from a class $S$ to a class $T$, a call to a method in $S$ gives rise to *implication constraints* of the form $\alpha = K \rightarrow c$. Here, $\alpha$ is a constraint variable, $K$ is a type, and $c$ is an unconditional constraint that must be satisfied if the condition holds. For example, consider the call `s5.v2.elements()` on line 45, which can be rewritten to an expression `s5.v2.iterator()` (see Figure 8). The implication constraints (x) and (xi) in Figure 9 state that the type of the call expression `s5.v2.elements()` is `Enumeration` if the type of `v2` remains `Vector`, but becomes `Iterator` if the expression is rewritten to `s5.v2.iterator()`.

Solving systems of implication constraints may require backtracking. However, it is often possible to perform simplifications that eliminate the need for implications. As an example, consider the call `v2.addElement(o)` on line 30. If the type of `v2` remains `Vector`, we must constrain `o` to be a subtype of the formal parameter of `Vector.addElement()`, which can be expressed by the constraint: [ `v2` ]=`Vector` → [ `o` ]$\leq$[ *Param*(`0,Vector:addElement(Object)`) ]. Similarly, for the case where the type of `v2` becomes `ArrayList`, we have: [ `v2` ]=`ArrayList` → [ `o` ]$\leq$[ *Param*(`0,ArrayList:add(Object)`) ]. These constraints can be combined into a single unconditional constraint [`o`]$\leq$`Object` (constraint (viii)) because both *Param*-expressions evaluate to `Object`.

[8] These types are only used during constraint solving. In other words, they are never introduced in the refactored source code.

```
class Client {
  public static void main(String[] args){
    Stack s1 = new Stack();
    s1.push(new Integer(1));
    Stack s2 = new Stack();
    s2.push(new Float(2.2));
    s2.push(new Float(3.3));
    s1.moveFrom(s2);
    s2.moveTo(s1);
    Stack.print(s2);
    Vector v1 = new Vector();
    while (!s1.isEmpty()){
      Number n = (Number)s1.pop();
      v1.add(n);
    }
    JFrame frame = new JFrame();
    frame.setTitle("Example");
    frame.setSize(300, 100);
    JTree tree = new JTree(v1);
    frame.add(tree, BorderLayout.CENTER);
    frame.setVisible(true);
  }
}
```

```
class Stack {
  private ArrayList v2;
  public Stack(){
    v2 = new ArrayList();
  }
  public void push(Object o){
    v2.add(o);
  }
  public void moveFrom(Stack s3){
    this.push(s3.pop());
  }
  public void moveTo(Stack s4){
    s4.push(this.pop());
  }
  public Object pop(){
    return v2.remove(v2.size() - 1);
  }
  public boolean isEmpty(){
    return v2.isEmpty();
  }
  public static void print(Stack s5){
    Iterator e = s5.v2.iterator();
    while (e.hasNext())
      System.out.println(e.next());
  }
}
```

**Fig. 10.** The example program after the application of Replace Class refactoring

From constraints (i) and (iv) in Figure 9, it follows that [ A1 ]$\leq$[ v1 ]$\leq$Vector, implying that the type A1 must remain Vector. However, the typing [ A2 ] $\leftarrow$ ArrayList, [ v2 ] $\leftarrow$ ArrayList satisfies the constraint system, indicating that allocation site A2 can be migrated to ArrayList.

Producing the refactored source code requires keeping track of the choices made for implication constraints and consulting the migration specification to determine how expressions should be rewritten. The refactored source code for the example program is shown in Figure 10.

A few additional complicating factors exist. In order to preserve synchronization behavior, we rely on a simple escape analysis to determine whether Vectors may escape their thread. Vectors that do not escape are migrated to ArrayLists (if no constraints are violated). For escaping Vectors, we attempt a translation that introduces a synchronization wrapper (rule (2) of Figure 8). Hence, there are three alternatives for each Vector allocation site: it can remain a Vector, become an unwrapped ArrayList, or a wrapped ArrayList. Preserving the behavior of downcasts requires additional constraints [1].

## 6 Related Work

Opdyke [15, page 27–28] identified some of the invariants that refactorings must preserve. One of these, *Compatible Signatures in Member Function Redefinition*, states that overriding methods must have corresponding argument types and return types, corresponding to our constraints (8) and (9). Opdyke writes the following about the *Type-Safe Assignments* invariant: "The type of each expression assigned to a variable must be an instance of the variable's defined type, or an instance of one of its subtypes. This applies both to assignment statements and function calls". This corresponds to our constraints (1), (3), (12), and (14).

Fowler [7] presents a comprehensive classification of a large number of refactorings, which includes step-by-step directions on how to perform each of these manually. Many of the thorny issues are not addressed. E.g., in the case of Extract Interface, Fowler only instructs one to "Adjust client type declarations to use the interface", ignoring the fact that not all declarations can be updated.

Tokuda and Batory [23] discuss refactorings for manipulating design patterns including one called Substitute which "generalizes a relationship by replacing a subclass reference to that of its superclass". Tokuda and Batory point out that "This refactoring must be highly constrained because it does not always work". Our model can be used to add the proper precondition checking.

Halloran and Scherlis [11] present an informal algorithm for detecting over-specific variable declarations. This algorithm is similar in spirit to our Generalize Declared Type refactoring by taking into account the members accessed from a variable, as well as the variables to which it is assigned.

The Infer Type refactoring by Steimann *et al.* [20] lets a programmer select a given variable and determines or creates a minimal interface that can be used as the type for that variable. Steimann et al. only present their type inference algorithm informally, but their constraints appear similar to those presented in Section 2. In more recent work, Steimann and Mayer [19] observe that the repeated use of Infer Type may produce suboptimal results (e.g., the creation of many similar types). Their Type Access Analyzer performs a global analysis to create a lattice that can be used as the basis for extracting supertypes, changing the types of declarations, merging structurally identical supertypes, etc.

The KABA tool [21,18] generates refactoring proposals for Java applications (e.g., indications that a class can be split, or that a member can be moved). In this work, type constraints record relationships between variables and members that must be preserved. From these type constraints, a binary relation between classes and members is constructed that encodes precisely the members that must be visible in each object. Concept analysis is used to generated a concept lattice from this relation, from which refactoring proposals are generated.

Duggan's approach for parameterizing classes [6] predates Java generics, and his PolyJava language is incompatible with Java in several respects (e.g., the treatment of raw types and arrays, no support for wildcards). Unlike our approach, Duggan's takes a class as its input and relies on usage information to generate constraints that relate the types of otherwise unrelated declarations. If usage information is incomplete or unavailable, too many type parameters may be inferred. To our knowledge, Duggan's work was never fully implemented.

Donovan and Ernst [4] present solutions to both the parameterization and the instantiation problems. For parameterization, a dataflow analysis is applied to each class to infer as many type parameters as are needed to ensure type-correctness. Then, type constraints are generated to infer how to instantiate occurrences of parameterized classes. Donovan and Ernst report that "often the class is over-generalized", i.e., too many type parameters are inferred. Donovan and Ernst's work predates Java generics (arrays of parameterized types are inferred, which are not allowed in Java) and was never fully implemented.

Donovan et al. [5] present a solution to the instantiation problem based on a context-sensitive pointer analysis. Their approach uses "guarded" constraints that are conditional on the rawness of a particular declaration, and that require a (limited) form of backtracking, similar to the implication constraints used in Section 5. Our solution is more scalable than Donovan's because it requires neither context-sensitive analysis nor backtracking, and more general because it is capable of inferring precise generic supertypes for subtypes of generic classes. Moreover, as Donovan's work predates Java 1.5, their refactoring tool does not consider wildcard types and supports arrays of generic types (now disallowed).

Von Dincklage and Diwan [25] present a solution to both the parameterization problem and the instantiation problem based on type constraints. Their Ilwith tool initially creates one type parameter per declaration, and then uses heuristics to merge type parameters. While the successful parameterization of several classes from the Java standard collections is reported, some of the inferred method signatures differ from those in the Java 1.5 libraries. It also appears that program behavior may be changed because constraints for overriding relationships between methods are missing. As a practical matter, Ilwith does not actually rewrite source code, but merely prints method signatures without providing details on how method *bodies* should be transformed.

## 7 Conclusion

An important category of refactorings is concerned with manipulating types and class hierarchies. For these refactorings, type constraints are an excellent basis for checking preconditions and computing source code modifications. We have discussed refactorings for generalization, for the introduction of generics, and for performing migrations between similar classes, using slight variations on a common type constraint formalism. All of our refactorings have been implemented in Eclipse, and several refactorings in the standard Eclipse distribution are based on our research. A detailed evaluation of the performance and effectiveness of our refactorings can be found in our earlier papers [8,1,13].

## Acknowledgments

The contents of this paper are based on the author's joint research with Ittai Balaban, Dirk Bäumer, Julian Dolby, Michael Ernst, Robert Fuhrer, Markus Keller, Adam Kieżun, and Bjorn De Sutter. Ittai Balaban, Adam Kieżun, and Jan Vitek provided valuable comments on drafts of this paper.

## References

1. Balaban, I., Tip, F., Fuhrer, R.: Refactoring support for class library migration. In: Proc. OOPSLA, pp. 265–279 (2005)
2. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, London, UK (2000)

3. De Sutter, B., Tip, F., Dolby, J.: Customization of Java library classes using type constraints and profile information. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 585–610. Springer, Heidelberg (2004)
4. Donovan, A., Ernst, M.: Inference of generic types in Java. Tech. Rep. MIT/LCS/TR-889, MIT (March 2003)
5. Donovan, A., Kieżun, A., Tschantz, M., Ernst, M.: Converting Java programs to use generic libraries. In: Proc. OOPSLA, pp. 15–34 (2004)
6. Duggan, D.: Modular type-based reverse engineering of parameterized types in Java code. In: Proc. OOPSLA, pp. 97–113 (1999)
7. Fowler, M.: Refactoring. In: Improving the Design of Existing Code, Addison-Wesley, London, UK (1999)
8. Fuhrer, R., Tip, F., Kieżun, A., Dolby, J., Keller, M.: Efficiently refactoring Java applications to use generic libraries. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 71–96. Springer, Heidelberg (2005)
9. Griswold, W.G.: Program Restructuring as an Aid to Software Maintenance. PhD thesis, University of Washington, Technical Report 91-08-04 (1991)
10. Griswold, W.G., Notkin, D.: Automated assistance for program restructuring. ACM Trans. Softw. Eng. Methodol. 2(3), 228–269 (1993)
11. Halloran, T.J., Scherlis, W.L.: Models of Thumb: Assuring best practice source code in large Java software systems. Tech. Rep. Fluid Project, School of Computer Science/ISRI, Carnegie Mellon University (September 2002)
12. Kerievsky, J.: Refactoring to Patterns. Addison-Wesley (2004)
13. Kieżun, A., Ernst, M., Tip, F., Fuhrer, R.: Refactoring for parameterizing Java classes. In: Proc. ICSE, pp. 437–446 (2007)
14. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. on Softw. Eng. 30(2), 126–139 (2004)
15. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. PhD thesis, University Of Illinois at Urbana-Champaign (1992)
16. Opdyke, W.F., Johnson, R.E.: Creating abstract superclasses by refactoring. In: The ACM 1993 Computer Science Conf. (CSC'93), February 1993, pp. 66–73 (1993)
17. Palsberg, J., Schwartzbach, M.: Object-Oriented Type Systems. John Wiley & Sons, West Sussex, England (1993)
18. Snelting, G., Tip, F.: Understanding class hierarchies using concept analysis. In: ACM Trans. on Programming Languages and Systems, May 2000, pp. 540–582 (2000)
19. Steimann, F., Mayer, P.: Type access analysis: Towards informed interface design. In: Proc. TOOLS Europe (to appear, 2007)
20. Steimann, F., Mayer, P., Meißner, A.: Decoupling classes with inferred interfaces. In: Preneel, B., Tavares, S. (eds.) SAC 2005. LNCS, vol. 3897, pp. 1404–1408. Springer, Heidelberg (2006)
21. Streckenbach, M., Snelting, G.: Refactoring class hierarchies with KABA. In: Proc. OOPSLA, pp. 315–330 (2004)
22. Tip, F., Kieżun, A., Bäumer, D.: Refactoring for generalization using type constraints. In: Proc. OOPSLA, pp. 13–26 (2003)
23. Tokuda, L., Batory, D.: Evolving object-oriented designs with refactorings. Kluwer Journal of Automated Software Engineering, 89–120 (August 2001)
24. Torgersen, M., Plesner Hansen, C., Ernst, E., von der Ahé, P., Bracha, G., Gafter, N.: Adding wildcards to the Java programming language. In: Proc. of the 2004 ACM symposium on Applied computing, pp. 1289–1296 (2004)
25. von Dincklage, D., Diwan, A.: Converting Java classes to use generics. In: Proc. OOPSLA, pp. 1–14 (2004)

# Programming Language Design and Analysis Motivated by Hardware Evolution
## (Invited Presentation)

Alan Mycroft

Computer Laboratory, University of Cambridge
William Gates Building, JJ Thomson Avenue,
Cambridge CB3 0FD, UK
`http://www.cl.cam.ac.uk/users/am`

**Abstract.** Silicon chip design has passed a threshold whereby exponentially increasing transistor density (Moore's Law) no longer translates into increased processing power for single-processor architectures. Moore's Law now expresses itself as an exponentially increasing number of processing cores per fixed-size chip.

We survey this process and its implications on programming language design and static analysis. Particular aspects addressed include the reduced reliability of ever-smaller components, the problems of physical distribution of programs and the growing problems of providing shared memory.

## 1 Hardware Background

Twenty years ago (1985 to be more precise) it was all so easy—processors and *matching* implementation languages were straightforward. The 5-stage pipeline of MIPS or SPARC was well established, the 80386 meant that the x86 architecture was now also 32-bit, and memory (DRAM) took 1–2 cycles to access. Moreover ANSI were in the process of standardising C which provided a near-perfect match to these architectures.

- Each primitive operation in C roughly[1] corresponded to one machine operator and took unit time.
- Virtual memory complicated the picture, but we largely took the view this was an "operating systems" rather than "application programmer" problem.
- C provided a convenient treaty point telling the programmer what the language did and did not guarantee; and a compiler could safely optimise based on this information. Classical dataflow optimisations (e.g. register allocation and global common sub-expression elimination) became common. A GCC port became a 'tick list requirement' for a new processor.

[1] The main wart was that a `struct` of arbitrary size could be copied with an innocent-looking '`=`' assignment.

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 18–33, 2007.

Moore's Law (the self-fulfilling guide that the number of transistors per unit area doubles every 18 months) continued to apply: a typical 1985 processor had a feature size of 1.5 $\mu$m, today's use 65 nm. The reduction in component size caused consequent (so called 'scaling') changes: speed increased and voltage was reduced. However, the *power* dissipated by a typical 2 cm$^2$ chip continued to increase—we saw this in the ever-increasing heat sinks on CPUs. Around 2005 it became clear that Moore's law would not continue to apply sensibly to x86 class processors. Going faster just dissipates too much power: the power needed to distribute a synchronous clock increases with clock speed—a typical uni-processor x86-style processor could spend 30% of its power merely doing this. Equally there are speed-of-light issues: even *light in vacuo* takes 100 ps ($\equiv$ 10 GHz) for the round trip across a 15 mm chip; real transistors driving real capacitive wires take far longer. For example the ITRS[2] works on the basis that the delay down 1 mm of copper on-chip wire (111 ps in 2006) will rise to 977 ps by 2013; this represents a cross-chip round-trip of nearly 30 ns—or 75 clock cycles even at a pedestrian 2.5 GHz—on a 15 mm chip. Of course, off chip-access to external memory will be far worse!

However, while Moore's Law cannot continue forever[3] it is still very much active: the architectural design space merely changed to multi-core processors. Instead of fighting technology to build a 5 GHz Pentium we build two or four 2.4 GHz processors ('cores') on a single chip and deem them 'better'. The individual cores shrink, and their largely independent nature means that we need to worry less about cross-the-whole-chip delays. However, making a four-core 2.4 GHz processor be more useful than a single-core 3 GHz processor requires significant program modification (either by programmer or by program analysis and optimisation); this point is a central issue to this paper and we return to it later.

Slightly surprisingly, over the past 20 years, the size of a typical chip has not changed significantly (making a chip much bigger tends to cause an unacceptable increase in manufacturing defects and costs) merely the density of components on it. This is one source of non-uniformity of scaling—and such non-uniformity may favour one architecture over another.

The current state of commercial research-art is Intel's 2006 announcement of their Tera-scale [11] Research Prototype Chips (2.75 cm$^2$, operating at 3.1 GHz). Rattner (Intel's CTO) is quoted[4] as saying:

> "...this chip's design consists of 80 tiles laid out in an 8x10 block array. Each tile includes a small core, or compute element, with a simple instruction set for processing floating-point data, ... . The tile also includes a router connecting the core to an on-chip network that links all the cores to each other and gives them access to memory.
>
> "The second major innovation is a 20 megabyte SRAM memory chip that is stacked on and bonded to the processor die. Stacking the die

[2] International Technology Roadmap for Semiconductors, `www.itrs.net`

[3] It is hard to see how a computing device can be smaller than an atom.

[4] `http://www.intel.com/pressroom/archive/releases/20060926corp_b.htm`

> makes possible thousands of interconnects and provides more than a terabyte-per-second of bandwidth between memory and the cores.”

MIT has been a major player in more academic consideration of processor designs which can be used to *tile* a chip. The RAW processor [21] made on-chip latencies visible to the assembly code processor to give predictable behaviour; the recent SCALE processor [2] addresses similar aims: “The Scale project is developing a new all-purpose programmable computing architecture for future system designs. Scale provides efficient support for all kinds of parallelism including data, thread, and instruction-level parallelism.”

The RAMP (Research Accelerator for Multiple Processors) consortium describe [24] an FPGA emulator for a range of new processors, and Asanovic et al. [2] summarise the “Landscape of Parallel Computing Research” from both hardware and software archetype (so-called ‘dwarfs’) perspectives.

### 1.1 Hidden Architectural Changes

The evolution, and particularly speed increase, from early single-chip CPUs to modern processors has not happened merely as a result of technology scaling. Much of the speed increase has been achieved (at significant cost in power dissipation) by spending additional transistors on components such as branch-prediction units, multiple-issue units and caches to compensate for non-uniformities in scaling.

The original RISC (‘Reduced Instruction Set Computer’) design philosophy of throwing away rarely-used instructions to allow faster execution of common instructions became re-written to a revisionist form “make each transistor pay its way in performance terms”. Certainly modern processors, particularly the x86, are hardly ‘reduced’ however, they do largely conform to this revised view. Thus, while originally it might have been seen as RISC (say) “to remove a division instruction limiting the critical timing path to allow the clock-speed to be increased”, later this came to be seen as “re-allocating transistors which have little overall performance effect (e.g. division) to rôles which have greater performance impact (e.g. pipeline, caches, branch prediction hardware, etc)”.

Another effect is that speed scaling has happened at very different rates. Processor speeds have increased rather faster than off-chip DRAM speeds. Many programmers are unaware that reading main memory on a typical PC takes *hundreds* of processor cycles. Data caches hide this effect for software which behaves ‘nicely’ which means not accessing memory as randomly as the acronym RAM would suggest. (Interestingly, modern cache designs seem to be starting to fail on the “make each transistor pay its way in performance terms” measure. Caches often exceed 50% of a chip area, but recent measurements show that typically on SPEC benchmarks that 80% of their content is dead data).

The result is that the performance of the modern processors depends far more critically on the exact instruction mix being executed than was historically the case; gone are the days when a load took 1–2 cycles.

Addressing such issues of timing is also a source of programming language design and program analysis interest. However it may be that this problem might diminish if, once multi-core is fully accepted, the fashion moved towards a greater number of simpler (and hence more predictable) processors rather than fewer complex processors.

## 1.2 Other Hardware Effects

As mentioned above, technology scaling reduces the size of components and increases switching speed even though voltage is also reduced. Propagating a signal over a wire whose length is in terms of feature size scales roughly (but not quite as well) as the switching speed of components. However, propagating a signal over a wire whose length is in terms of chip size gets exponentially worse in terms of gate delays. This can be counteracted by using larger driver transistors, or more effectively by placing additional gates (buffering) along the path. But doing computation within these buffers adds very little to the total delay. Two slogans can summarise this situation:

- (the expanding universe): communication with neighbouring components scales well, but every year more components appear between you and the edge of the chip and communicating with these requires either exponentially more delay, or exponentially bigger driver transistors. Thus local computation wins over communication.
- (use gzip at sender and gunzip at receiver); it is worth spending increasing amounts of computation to reduce the number of bits being sent because this allows greater delays between each transition in turn which allows smaller transistors to drive the wire.

Once a long wire has gates on it, then we may as well do something useful with them. This fortuitously overlaps the question of how we enable the many processors on a multi-core chip to communicate with one another. The answer to use is some form of *on-chip network*. This can be fast (if big driver transistors are used) and compare favourably with communication achieved by random point-to-point links.

Reducing the feature size of a chip generally requires its operating voltage to be reduced. However, this reduces noise margins and also increases the ‘leakage current’ (the classical CMOS assumption is that transistors on a chip only consume one unit of energy when switching); below 65 nm or so leakage current can exceed power consumption due to computation. This gives two effects:

- it is beneficial to turn off the power supply to areas of the chip which are not currently active;
- the reduced noise margin increases the error rate on longer wires—sometimes it is only half-jokingly suggested that TCP/IP might be a good protocol for on-chip networks.

A further effect of feature size reduction is that as transistors become smaller, their state (on/off) is expressed as a small number of electrons. This makes the transistors more susceptible to damage by charged particles e.g. cosmic rays, natural radiation. Charged particles may permanently damage a device, but it is far more common for it to produce a *transient event*, also known as Single Event Transient (SET). For example the electrons liberated by a charged particle may enable a transistor whose input demands it be off (non-conducting) to conduct for a short period of time (600 ps is quoted). Some DRAM memory chips ('ECC') are equipped with redundant memory cells and error-detecting or error-correcting codes generated during write cycles and applied during read cycles; similarly aerospace often uses multiple independent devices and majority voting circuits. It is notable that a standard aircraft 'autopilot' function is implemented using five 80386 processors; the larger feature size of the 80386 makes it less susceptible to SET events, and the multiple processors give significant redundancy.

### 1.3 Summary: Hardware Evolution Effects on Programming

The hardware changes discussed above can be summarised in the following points:

- computation is increasingly cheap compared to communication;
- making effective use of the hardware requires more and more parallelism;
- the idea of global shared memory (and with it notions like semaphores and locking) is becoming less sustainable; on-chip networks increasingly connect components on chip;
- to keep chips cool enough we may have to move CPU-intensive processes around, and (because of leakage current) to disable parts of the chip when they are not being used;
- because of smaller feature sizes, transient hardware errors (leading to errors in data) will become more common.

One additional effect is:

- the time taken for a computation has become much less predictable due to the complexity of uni-processor designs, caches and the like. However, this may reduce in the future were individual processors to become simpler and the idea of a uniform memory space were to be less of a programming language assumption.

The rest of the paper considers programming language consequences.

## 2 Programming Language Mismatch to Hardware

Most current mainstream programming languages are pretty much stuck in the 1985 model. It is true that C has been superseded by the Object-Oriented paradigm (C++, Java, C#) and this is certainly useful for software engineering as it facilitates larger systems being created by programming teams. However, in many ways very little has changed from C.

Perhaps the critical observation is that the current OO fashion coincided with the period of steady *evolution* of the (single-processor) x86 architecture. The current pressures for *revolutionary* change may (and I argue should) lead to matching language change. Let us examine some particular problems with the OO paradigm (many are inherited from C).

Consider a function (or method) `foo` which takes an element of `class C` as a parameter. In C++, merely to *declare* `foo` (e.g. as part of an interface specification before any code has been written) we have to choose between three possible declarations:

```
extern void foo(C x);
extern void foo(C *x);
extern void foo(C &x);
```

The first one says call-by-value, the last two provide syntactic variants on call-by-reference. While on a single-processor architecture we might discuss these in terms of minor efficiency considerations (e.g. whether `class C` is small enough that its copying involved in call-by-value makes up for additional memory accesses to its fields implied by call-by-reference), on a multi-core architecture the difference is much more fundamental. If call-by-reference is used then the caller of `foo` and the body of `foo` must by executed on processors both of which have access to the memory pointed to by `x`. While call-by-value might therefore seem attractive, when used without care it breaks typical OO assumptions ('object identity'). E.g. toggling one switch twice may well have a different effect from toggling both a switch and its clone.

It might be suggested that Java avoids this issue; however it avoids it in the way that in '1984' Orwell's totalitarian government encourages 'Newspeak' to avoid thought-crime by making it inexpressible in the language. In Java all calls are by reference; therefore caller and callee must execute on processors sharing access to passed data. Call-by-value is at best clumsily expressed via remote method invocation (RMI); moreover, any suggestion that RMI can be introduced "where necessary" to distribute a Java system is doomed to fail due to the very different syntax and semantics of local and remote method invocation.

Incidentally, the `restrict` qualifier found in C99 does not help significantly here in that it keeps the assumption of global shared memory, but merely allows the compiler to make various (non-)aliasing assumptions.

We will return to this topic below, but I would like to argue that it is inappropriate to have interface specification syntax which places restrictions ('early binding') on physical distribution of methods; one suggestion is that of a C++ variant declaration

```
extern void foo(C @x);
```

with meaning "I've not yet decided whether to pass `x` by value or by reference, so fault my program if I do an operation which would have differing semantics under the two regimes."

This would allow *late binding* of physical distribution: all uses of '`@`' can be treated as copy (if caller and callee do not share a memory space at reasonable cost) or by alias (if caller and callee execute on the same address space).

### 2.1 What Should This Community Learn?

We will return to topics below in more detail, but we list them here to motivate the topics we choose below.

Processor developments expose the fact that C-like languages do not capture important properties of system design and implementation for such architectures. Important issues which we might want to expose to aid writing software for such architectures include:

- late binding of physical distribution;
- more expressive, but concise interface specifications;
- systematic treatment of transient errors.

### 2.2 Why Not Stick with C++/Java/C#?

For many traditional applications this will suffice, e.g. editors, compilers, spreadsheets and the like will continue to work well on a single core of a multi-core processor. But current and future applications (e.g. weather forecasting, sophisticated modelling[5]) will continue to demand effective exploitation of hardware—and this means exploiting concurrency.

Remember also that one particular challenge will be to execute programs written for packages—such as Matlab—effectively.

## 3 Programming Language Design *or* Program Analysis

A great deal of work in this community concerns program *analysis*. While I have personally worked on program analysis, and continue to believe in it for local analysis, I now have significant doubts as regards whole-program (inter-module) analyses and their software engineering impact. The problem is that of *discontinuity*: such a large-scale analysis may enable some optimisation which gains a large speed-up, for example by enabling physical distribution. However, a programmer may then make a seemingly minor change (perhaps one of a long sequence of changes) to a program only to find that it now runs many times slower. This real problem is that it is hard to formulate what the programmer 'did wrong' and to enable understanding of how to make a similar change without suffering this penalty.

Type systems and properties expressed by type-like systems seem to provide a better answer: properties of interest are then explicitly represented within a

[5] One might wonder how *our* community might exploit large-scale concurrency. Do our programs match any of the software dwarfs of Asanovic et al. [2]? Or will we merely continue to use use one core of our multi-core chips as we do at the moment?

programming language at interface points. For example, the interface to a procedure may express the property that its parameter will be consumed (logically de-allocated); then callers of the procedure can check that no use is made of the parameter after the call. This means that violating important assumptions will result in errors which have human-comprehensible explanations.

Of course, there is no problem with *local* inference—a tasteful mechanism always avoids pointless specification—the advantage is that programmer-specified invariants can be used to anchor local reasoning and to break down a global analysis into many smaller independent analyses.

For various aspects of multi-core programming (e.g. running two blocks of code concurrently), it is important to know whether two pointers may alias. Much important work has been done on *alias analysis* (which is undecidable in theory and for which achieving good approximations is problematic in practice). However, we still lack designs for programming languages in which aliasing information can be expressed within the language rather than as a mere analysis.

This situation can be compared with the two ways of adding types to a dynamically-typed language. Method 1 is to analyse the program determining which variables have known type (and optimising their accesses) and which have to adopt a fall-back 'could be anything' treatment. Method 2 is to add a mandatory type system for the language which allows all variable accesses to be optimised and which rejects as few programs as possible. Lisp (with Soft Typing) and ML might serve as good templates here.

An implementation language in which programmer knowledge of known aliasing (and non-aliasing) can be expressed in interface specifications succinctly, and acceptably to an ordinary programmer, would be a particular success here.

## 4 Programming Languages: High-Level and Implementation Level

Some might be surprised at my focus on C in the introduction, when there were arguably many more 'interesting' language features being explored in 1985. I focused on C because it corresponded directly to hardware features in 1985 (it was a good *implementation* language), and indeed could (and did) usefully serve as an intermediate language for compilers from higher-level languages.

The same motivation was also present in the design of Occam [10] which was a well-matched implementation language for an early multi-core computer[6]—the 'transputer'.[7] Occam was modelled on Hoare's CSP so naturally supported message passing (separate transputers had no shared memory), and represented concurrency directly. Sadly, in many ways it was ahead of its time, and we would

[6] The 'cores' here were actually separate chips containing 16-bit (later 32-bit) processors with their own memory and four fast I/O ports (e.g. with nearest neighbours on a rectangular grid).

[7] `http://en.wikipedia.org/wiki/INMOS_transputer` seems the best reference nowadays.

have perhaps been in a better position with respect to higher-level language for multi-core had more work been done then on novel higher-level features to compile to Occam!

My feeling here is that we need both sorts of language; we need a good implementation language which is well-matched to hardware, and this exhibits the 'sharp end' of many challenges. However, high-level languages may also need to change somewhat, and this is taken up in the next section with "pointers considered harmful" and Section 5.7 explores the desire to have late binding of store-versus-recompute decisions. Some form of structuring beyond "everything is an object (or value) which can be used anywhere at any time" appears likely to be useful.

## 5 Some Interesting Directions

This section discusses work or possible future projects (inevitably biased towards my own interests) which could be useful in addressing the mismatch between current languages and future hardware.

### 5.1 (Simple) Pointers Considered Harmful

We have already identified how the ubiquitous use of call-by-reference for objects causes two forms of problems. Firstly in practice it becomes almost impossible to determine whether two object references point to the same object or not, this inhibits parallelisation since code as simple as

```
for (i=0; i<NCHANS; i++) process_channel(i);
```

is often not parallelisable because of the possibility of aliasing of some reference in `process_channel(0)` with a reference in `process_channel(1)`. Secondly, pointers inhibit physical distribution in situations where memory access is not uniform, e.g. if a caller and callee are on different physical processors then we need to ensure that any call-by-reference parameter lives in memory accessible to both—and this may not exist or may be much slower to access than fast local memory. C99's `restrict` qualifier can sometimes help with the former problem but not with the latter.

Shape Analysis [16], Uniqueness types [13] and Ownership Types [3] provide some purchase on this problem. They identify a similar theme: unrestricted pointers are too powerful. In many ways they are like the unrestricted use of labels and `goto`s which Dijkstra railed against in "`goto` considered harmful". Given that a pointer to local data on one processor is not necessarily even valid on another processor, we need some way to tame pointers. This can happen in more than one way: either we wish to control the number of aliases to a given object, or we wish to ascribe an address space to a pointer, so that dereferencing a pointer is only valid on processors which have a capability to do so.

In our work on PacLang [4] we showed how a quasi-linear[8] type system allows one to write code naturally in which an object could move between processors with compile time checking of linearity assumptions. A purely linear type system tends to require rather uncomfortable passing of values back and forth; a quasi-linear system enhances this with limited-lifetime second-class pointers which make local call-by-reference possible in a natural programming style. It turns out that linear (or quasi-linear) knowledge of pointers helps [5] in refactoring code from sequential to concurrent by telling a compiler that certain aliasing—which would lead to a race condition—cannot happen. A particular noteworthy point was the concept of an "Architecture Mapping Script (AMS)" which specified architecture details so that late binding of processes to processors could be achieved.

Microsoft's Singularity OS (Fähndrich et al. [6]) project further develops this idea to a message-passing operating system—linear data buffers (allocated in `ExHeap`) can be transferred from process to process by merely passing a pointer since linearity ensures that the sender no longer has access to the data after transfer.

Region-based type systems [19] distinguish pointer types with the region (think 'address space') into which they may legitimately point. They provide a good basis for providing syntax to describe situations in which a pointer to local memory in one processor is being passed via a second processor (on which it is not valid to dereference it) onto a third which can dereference it. However, as Fähndrich et al. observe, the original lexical nesting structure of regions cannot be retained as-is.

### 5.2 The Actor Model

In the Actor Model of computation, all inter-process communication is achieved by message passing; actors only access disjoint memory. Given that architectural developments mean that communication is becoming the dominant cost rather than computation, actor-based languages are appealing for their explicit representation of non-local communication. A notable commercial example is that of Erlang [1].

### 5.3 Theoretical Models of Restricted Re-use

We can see all the above models as attempts to control how and when value might validly be accessed—contrast this with traditional shared memory in which any value may be accessed at any time so long as it has not been overwritten.

While models based on linear types have been mentioned several times above (Wadler [20] is the seminal explanation of this), Separation Logic [14] has recently

[8] A linear type system requires each object to have a single active pointer and when that pointer is assigned or passed to a function then it may no longer be used to reference the object; only the new copy may be so used. In C++ a dynamic version of this concept is enshrined as the `auto_ptr` class and invalidation of old pointers is achieved by overriding the assignment operator.

attracted rapidly growing interest. Separation logic at its simplest expresses assertions that an address space is split into two or more disjoint areas—not only can this model the sort of situations we have seen above, but it can also model dynamic change of ownership and shared-memory systems very effectively.

Both linear types and separation logic provide mechanisms for describing values which are not freely accessible from the whole program. However, they describe overlapping rather than identical phenomena and a formal connection between them would be highly desirable for inspiring work on concise programming language representations of restricted re-use.

### 5.4 More on Interface Specifications

We have already seen that providing interface specification on pointers can provide compile-time to programs to enable them to be better mapped onto multicore hardware.

However, there are other ways in which interfaces are often inexpressive and we give a hardware example.[9] Consider the Verilog encoding of a two-stage shift register: the input byte on '`in`' appears on the output '`out`' two clock ticks later:

```
module two_stage (in, out, clock);
   input [7:0] in;
   output [7:0] out;
   input clock;

   reg [7:0] state1;
   reg [7:0] state2;
   assign out = state2;         // or out = state1
   always @(posedge clock)
         begin
            state1 <= in;
            state2 <= state1;
         end
endmodule
```

In this example the module specification contains the classical programming language knowledge that `in`, `out`, and `clock` are wires of given width and which of them are outputs. However, suppose we change the commented line to

```
   assign out = state1;
```

then the code behaves as a one-stage delay instead. In an ideal world, we would like this timing information, or indeed information that one signal is only valid

[9] It is admitted that HDLs (Hardware Description Languages) are moving towards using FIFO "channels" to add flexibility in the time domain to ease this sort of problem, particularly for crossing clock domain boundaries, but the example given illustrates another way in which classical programming language "names and types" interface formalism can be seen as lacking.

after another goes high, to be part of the type information in the module specification so that if part of a circuit is retimed—say to equalise computation delays—then type-checking errors can be raised for places in the code which have not taken this retiming into account.

Does such information have a place in future programming languages?

### 5.5 Fractals in Programming and Architecture

Rent's Rule (Stroobandt [17] gives a good overview) enshrines the empirical observation that the number of pins $T$ on a chip tends to follow a power law $T = T_0 g^p$ where $g$ is the number of internal components (gates) and $T_0$ and $p$ are constants. Donath noted that Rent's Rule could also be be used to estimate wire-length distribution in VLSI chips.

Power laws tend to suggest that there is an underlying self-similarity (also known as the system being fractal). Of course, the very basis of top-down engineering (software or otherwise) is that each component is hierarchically built of a number of smaller components. This also is self-similar.

It is intriguing to consider whether these two observations could be exploited to improve our understanding of how to map complex systems to hardware. With the exception of references to global memory (heap-allocated data structures—recall the "pointers considered harmful" slogan above) data flow follows design decomposition which is very encouraging. Do global-memory-free programs have a better mapping to hardware? Can such programs be expressive enough?

Of course, global memory breaks this assumption because it provides a way to move data from any part of the system to another in a relatively uncontrolled manner; in general memory access requires some form of serialisation which slows down processing elements. One question is whether designs like Intel's 'Tera-scale' (bonding the memory directly on top of the processor die essentially exploits a scale-free short-path access in the third dimension, see Section 1) will provide enough bandwidth to (and cache-coherency of!) global memory so that shared-memory models are still valid or whether the concept of global memory is still ultimately problematical.

### 5.6 Limits for Speed-Ups

In seminal work Wall [23] analyses instruction traces for various benchmarks including an early version of the SPEC[10] benchmark suite. He calculated limits to speed-up based on *instruction-level* parallelism under various models of behaviour of caches and branch prediction. Redux [12] constructed a more abstract "Dynamic Data Flow Graph (DDFG)" from a computation which reflected merely the computations necessary to compute the result. The depth of the DDFG therefore represents the computation given unbounded parallelism (not necessarily limited to instruction-level parallelism).

An interesting research direction might be to explore more whether DDFGs for various benchmarks might fit (or be made to fit by source-level adjustment)

[10] `www.spec.org`

a world in which parallel computation is cheap, but communication is expensive. To what extent to DDFGs express fractal structure inherent in programs?

### 5.7 Store Versus Re-compute

Traditionally, the sequential nature of computers has generally made programmers aware that re-computation takes time and the way to solve this is to store values for later re-use. However, as memory becomes more distributed then local re-computation of some values (especially data structures in memory) may become cheaper than accessing remote memory.

We have already seen a small version of this in optimising compilers: if a small common sub-expression (e.g. `x+1`), which would normally held in a register, has to be spilled to memory, then it is better to *rematerialise* it (i.e. recompute it, thereby undoing the CSE optimisation) rather than accept the cost of a store followed by a re-load.

Again it would be desirable to to have language features which allow programs to be designed and developed more neutrally with respect to store versus recompute ('late binding on the store/recompute axis') than is currently the case. This would also facilitate porting to multiple architectures.

### 5.8 Opportunistic Concurrency and Related Techniques

While the status of global memory in multi-processor systems is unclear, there is much scope for examining alternatives to today's ubiquitous semaphore-based locking and unlocking mechanisms which can be expensive due both to the cost of memory synchronisation for atomic test-and-set or compare-and-swap instructions and also to the fact that programmers often find it easier to take a coarse-grain lock instead of reasoning about the correctness of fine-grain locking.

One of these is Software Transactional Memory (STM) [7], in which lock and unlock are replaced by an `atomic` block. Atomic blocks execute speculatively in that either they execute to completion without interference from other processes able to access given memory, or such interference is detected and the atomic block (repeatedly) re-tried. This idea is already familiar from databases. Software Transactional Memory represents an interesting mid-point between the notion of lockable global memory and the notion of message-passing.

Worth grouping with STM is Rundberg and Stenstrom's Data Dependence Speculation System [15] in which ambiguities in compile-time alias analysis are resolved at run time by speculatively executing threads concurrently but with a dynamic test which effectively suppresses writes occurring out of order and restarts the threads from a suitable point.

Lokhmotov et al. [9] describe Codeplay's *sieve* construct which side-steps many of the problems with alias analysis. A *sieve* block has writes with the block delayed until the block exit. This allows a programmer to express the absence of read-after-write dependencies (which often have to be assumed present due to inaccuracy of alias analysis); the effect is to allow C-like code to be optimised into a "DMA-in, process in parallel, then DMA-out" form.

Finally, there has been significant work on pre-fetch- or turbo-threads in which two versions of code are compiled. The second one is the normal code, and the first one is a cut-down version which is intended to execute in advance of the second one; it does reduced computation with no stores to global memory and all loads from local memory replaced by pre-fetch instructions (start load into cache but do not wait for result).

### 5.9 Programming Against Hazards and Transient Errors

This is a relatively new topic for our community. We probably are all aware that highly safety-critical (usually for aerospace applications) electronics is often duplicated and a majority-voting circuit avoids any single processor crashing (temporarily or even permanently) causing mission failure. Embedded systems programmers have for years used 'watchdog timers' to ensure that systems suffering temporary failure (e.g. an infinite loop caused by data being corrupted by a cosmic ray), can be restarted relatively quickly. The idea is that the 'watchdog' has to be sent a message periodically confirming the sender is still alive; if this does not happen then the watchdog (hardware) timer causes a system reset. Of course, there have been research groups focusing on overall system reliability for decades, but recently there has been several novel approaches which intersect our community's interests more directly.

Some work which springs to mind includes:

- Sarah Thompson's thesis [18] showing not only that hazards (narrow pulses on an otherwise clean signal or transition) can be modelled by abstract interpretation, but also that majority voting circuits are theoretically incapable of removing them without resorting to timing.
- Walker et al. describe lambda-zap [22] whose reduction both inserts faults and also duplicates computations to be resolved by majority voting. There is also a type system that guarantees well-typed programs can tolerate a single error.
- Hillston's PEPA [8] (Performance Evaluation Process Algebra) which can model quantitative aspects of systems, originally reaction rates, but hopefully also failure rates.

Finally, there is idea that performance and reliability can be traded, for example we might eventually require some form of redundant computation—perhaps merely at the hardware level—to give enough reliability for (say) a spreadsheet, but be tolerant of errors during some applications (e.g. rendering graphics for display where the human eye either would not notice errors or would ignore them).

## 6 Conclusion

We have seen that forces in hardware design are increasing the mismatch between traditional programming languages and future processor designs. I argue that *languages* have to evolve to take into account the new emphasis on concurrency and the reduced ability to view an object as a simple pointer—and more expressive interface specifications are pivotal. Program *analysis* techniques can provide inspiration for these future designs.

In Addition, there remains scope for language features designed to address issues directly concerning hardware; for example, how is unreliability best expressed? Can we improve the expressivity of hardware and software interfaces to document better their behaviour?

In the bigger picture, there is still much scope for higher-level language designs which encourage programmers to think in a way which naturally encodes effectively on coming architectures—and even for new architectural features corresponding to programming innovations. Can we return to the comfort of 1985 when implementation languages and computer architecture *matched*?

## Acknowledgements

Thanks are due to Simon Moore at Cambridge for many useful ideas and comments.

## References

1. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. Prentice Hall, Englewood Cliffs (1996)
2. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (December 18, 2006)
3. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA '98. Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, pp. 48–64. ACM Press, New York (1998)
4. Ennals, R., Sharp, R., Mycroft, A.: Linear types for packet processing. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 204–218. Springer, Heidelberg (2004)
5. Ennals, R., Sharp, R., Mycroft, A.: Task partitioning for multi-core network processors. In: Bodik, R. (ed.) CC 2005. LNCS, vol. 3443, pp. 76–90. Springer, Heidelberg (2005)
6. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R., Levi, S.: Language support for fast and reliable message-based communication in singularity OS. In: EuroSys '06. Proceedings of the 2006 EuroSys conference, New York, NY, USA, pp. 177–190. ACM Press, New York (2006)
7. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA '03. Proceedings of the 18th annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications, New York, NY, USA, vol. 38, pp. 388–402. ACM Press, New York (2003)
8. Hillston, J.: Tuning systems: from composition to performance The Needham Lecture. Comput. J. 48(4), 385–400 (2005)

9. Lokhmotov, A., Mycroft, A., Richards, A.: Delayed side-effects ease multi-core programming. In: Euro-Par 2007 Parallel Processing. LNCS, Springer, Heidelberg (2007)
10. INMOS Ltd. OCCAM Programming Manual. Prentice-Hall International, London (1984)
11. Intel Corporation. Terascale computing. `http://www.intel.com/research/platform/terascale/index.htm`
12. Nethercote, N., Mycroft, A.: Redux: A dynamic dataflow tracer. Electr. Notes Theor. Comput. Sci. 89(2) (2003)
13. Plasmeijer, M., van Eekelen, M.: Language report: concurrent Clean, Technical Report CSI-R9816, Computing Science Institute, University of Nijmegen, The Netherlands (1998)
14. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proc. IEEE Symposium on Logic in Computer Science, pp. 55–74. IEEE Computer Society Press, Los Alamitos (2002)
15. Rundberg, P., Stenstrom, P.: An All-Software Thread-Level Data Dependence Speculation System for Multiprocessors. The Journal of Instruction-Level Parallelism (1999)
16. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL, pp. 105–118 (1999)
17. Stroobandt, D.: Recent advances in system-level interconnect prediction. IEEE Circuits and Systems Society Newsletter. 11(4), 4–20 (2000) Available at `http://www.nd.edu/~stjoseph/newscas/`
18. Thompson, S.: On the application of program analysis and transformation to high reliability hardware. Technical Report UCAM-CL-TR-670 (PhD thesis), University of Cambridge, Computer Laboratory (.July 2006) Available at `http://www.cl.cam.ac.uk/techreports`
19. Tofte, M., Talpin, J.-P.: Region-based memory management. Information and Computation 132(2), 109–176 (1997)
20. Wadler, P.: Linear types can change the world. In: Broy, M., Jones, C.( eds.) IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, pp. 347–359 North Holland (1990)
21. Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S., Agarwal, A.: Baring it all to software: Raw machines. Computer 30(9), 86–93 (1997)
22. Walker, D., Mackey, L., Ligatti, J., Reis, G., August, D.: Static typing for a faulty lambda calculus. In: ACM International Conference on Functional Programming, Portland (September 2006)
23. Wall, D.W.: Limits of instruction-level parallelism. In: ASPLOS. Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System, New York, NY, vol. 26, pp. 176–189. ACM Press, New York (1991)
24. Wawrzynek, J., Oskin, M., Kozyrakis, C., Chiou, D., Patterson, D.A., Lu, S.-L., Hoe, J.C., Asanovic, K.: Ramp: a research accelerator for multiple processors. Technical Report UCB/EECS-2006-158, EECS Department, University of California, Berkeley (November 24 2006)

# A Compilation Model for Aspect-Oriented Polymorphically Typed Functional Languages

Kung Chen[1], Shu-Chun Weng[2], Meng Wang[3],
Siau-Cheng Khoo[4], and Chung-Hsin Chen[1]

[1] National Chengchi University
[2] National Taiwan University
[3] Oxford University
[4] National University of Singapore

**Abstract.** Introducing aspect orientation to a polymorphically typed functional language strengthens the importance of *type-scoped advices*; i.e., advices with their effects harnessed by type constraints. As types are typically treated as compile time entities, it is highly desirable to be able to perform *static weaving* to determine at compile time the *chaining* of type-scoped advices to their associated join points. In this paper, we describe a compilation model, as well as its implementation, that supports static type inference and static weaving of programs in an aspect-oriented polymorphically typed lazy functional language, AspectFun. We present a type-directed weaving scheme that coherently weaves type-scoped advices into the base program at compile time. We state the correctness of the static weaving with respect to the operational semantics of AspectFun. We also demonstrate how control-flow based pointcuts (such as `cflowbelow`) are compiled away, and highlight several type-directed optimization strategies that can improve the efficiency of woven code.

## 1 Introduction

Aspect-oriented programming (AOP) aims at modularizing concerns such as profiling and security that crosscut the components of a software system[8]. In AOP, a program consists of many functional modules and some *aspects* that encapsulate the crosscutting concerns. An aspect provides two specifications: A *pointcut*, comprising a set of functions, designates when and where to crosscut other modules; and an *advice*, which is a piece of code, that will be executed when a pointcut is reached. The complete program behaviour is derived by some novel ways of composing functional modules and aspects according to the specifications given within the aspects. This is called *weaving* in AOP. Weaving results in the behaviour of those functional modules impacted by aspects being modified accordingly.

The effect of an aspect on a group of functions can be controlled by introducing *bounded scope* to the aspect. Specifically, when the AOP paradigm is supported by a strongly-type polymorphic functional language, such as Haskell or ML, it is natural to limit the effect of an aspect on a function through declaration

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 34–51, 2007.

of the *argument type*. For instance, the code shown in Figure 1 defines three aspects named `n3`, `n4`, and `n5` respectively; it also defines a main/base program consisting of declarations of `f` and `h` and a main expression returning a triplet. These advices designate `h` as *pointcut*. They differ in the type constraints of their first arguments. While `n3` is triggered at all invocations of `h`, `n4` limits the scope of its impact through type scoping on its first argument; this is called a *type-scoped* advice. This means that execution of `n4` will be woven into only those invocations of `h` with arguments of list type. Lastly, the type-scoped advice `n5` will only be woven into those invocations of `h` with their arguments being strings.

*Example 1.*

```
// Aspects
n3@advice around {h} (arg) =
   proceed arg ;
   println "exiting from h" in
n4@advice around {h} (arg:[a]) =
   println "entering with a list";
   proceed arg in
n5@advice around {h} (arg:[Char]) =
   print "entering with ";
   println arg;
   proceed arg in
// Base program
h x = x in
f x = h x in (f "c", f [1], h [2])
```

```
// Execution trace
entering with a list
entering with c
exiting from h

entering with a list
exiting from h
entering with a list
exiting from h
```

**Fig. 1.** An Example of Aspect-oriented program written in AspectFun

As with other AOP, we use `proceed` as a special keyword which may be called inside the body of an *around* advice. It is bound to a function that represents "the rest of the computation at the advised function"; specifically, it enables the control to revert to the advised function (ie., `h`).

Using type-scoped aspects enable us to have customized, type-dependent tracing message. Note that *String* (a list of *Char*) is treated differently from ordinary lists. Assuming a textual order of advice triggering, the corresponding trace messages produced by executing the complete program is displayed to the right of the example code.

In the setting of strongly-type polymorphic functional languages, types are treated as compile-time entities. As their use in controlling advices can usually be determined at compile-time, it is desirable to perform *static weaving* of advices into base program at compile time to produce an integrated code without explicit declaration of aspects. As pointed out by Sereni and de Moor [13], the integrated woven code produced by static weaving can facilitate static analysis of aspect-oriented programs.

Despite its benefits, static weaving is never a trivial task, especially in the presence of type-scoped advices. Specifically, it is not always possible to determine *locally* at compile time if a particular advice should be woven. Consider

Example 1, from a syntactic viewpoint, function `h` can be called in the body of `f`. If we were to naively infer that the argument `x` to function `h` in the RHS of `f`'s definition is of polymorphic type, we would be tempted to conclude that (1) advice `n3` should be triggered at the call, and (2) advices `n4` and `n5` should not be called as its type-scope is less general than $a \rightarrow a$. As a result, only `n3` would be statically applied to the call to `h`.

Unfortunately, this approach would cause inconsistent behavior of `h` at runtime, as only the third trace message "`exiting from h`" would be printed. This would be incoherent because the invocations `(h [1])` (indirectly called from `(f [1])`) and `(h [2])` would exhibit different behaviors even though they would receive arguments of the same type.

Most of the work on aspect-oriented functional languages do not address this issue of static and yet coherent weaving. In AspectML [4] (*a.k.a* PolyAML [3]), dynamic type checking is employed to handle matching of type-scoped pointcuts; on the other hand, Aspectual Caml [10] takes a lexical approach which sacrifices coherence[1] for static weaving.

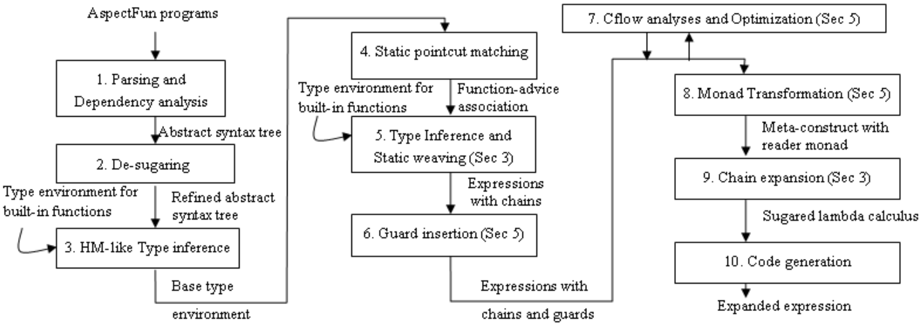

AspectFun programs
1. Parsing and Dependency analysis
Abstract syntax tree
2. De-sugaring
Type environment for built-in functions
Refined abstract syntax tree
3. HM-like Type inference
Base type environment
4. Static pointcut matching
Type environment for built-in functions
Function-advice association
5. Type Inference and Static weaving (Sec 3)
Expressions with chains
6. Guard insertion (Sec 5)
Expressions with chains and guards
7. Cflow analyses and Optimization (Sec 5)
8. Monad Transformation (Sec 5)
Meta-construct with reader monad
9. Chain expansion (Sec 3)
Sugared lambda calculus
10. Code generation
Expanded expression

**Fig. 2.** Compilation Model for AspectFun

In this paper, we present a compilation model for AspectFun that ensures static and coherent weaving. AspectFun is an aspect-oriented polymorphically typed functional language with lazy semantics. The overall compilation process is illustrated in Figure 2. Briefly, the model comprises the following three major steps: (1) Static type inference of an aspect-oriented program; (2) Type-directed static weaving to convert advices to functions and produce a piece of woven code; (3) Type-directed optimization of the woven code. In contrast with our earlier work [15], this compilation model extends our research in three dimensions:

1. Language features: We have included a suite of features to our aspect-oriented functional language, AspectFun. Presented in this paper are: *second-order*

[1] Our notion of coherence admits semantic equivalence among different invocations of a function with the same argument type. This is different from the coherence concept defined in qualified types [6] which states that different translations of an expression are semantically equivalent.

*advices*, complex pointcuts such as `cflowbelow`, and an operational semantics for AspectFun.
2. Algorithms: We have extended our type inference and static weaving strategy to handle the language extension.[2] We have formulated the correctness of static weaving *wrt.* the operational semantics of AspectFun, and provided a strategy for analysing and optimizing the use of `cflowbelow` pointcuts.
3. Systems: We have provided a complete implementation of our compilation model turning aspect-oriented functional programs into executable Haskell code.[3]

Under our compilation scheme, the program in Example 1 is first translated through static weaving to an expression in lambda-calculus with constants for execution. For presentation sake, the following result of static weaving is expressed using some meta-constructs:

```
n3 = \arg -> (proceed arg ; println "exiting from h") in
n4 = \arg -> (print "entering h with a list" ; proceed arg) in
n5 = \arg -> (print "entering h with " ; println arg; proceed arg) in
h x = x in
f dh x = dh x in (f <h,{n3,n4,n5}> "c", f <h,{n3,n4}> [1], <h,{n3,n4}> [2])
```

Note that all advice declarations are translated into functions and are woven in. A *meta*-construct $\langle \_ \,, \{\ldots\}\rangle$, called *chain expression*, is used to express the chaining of advices and advised functions. For instance, $\langle \mathtt{h}\ , \{\mathtt{n3}, \mathtt{n4}\}\rangle$ denotes the chaining of advices `n3` and `n4` to advised function `h`. In the above example, the two invocations of `h`, with integer-list arguments, in the original aspect program have been translated to invocations of the chain expression $\langle \mathtt{h}\ , \{\mathtt{n3}, \mathtt{n4}\}\rangle$. This shows that our weaver respects the coherence property.

All the technically challenging stages in the compilation process are explained in detail – in their respective sections – in the rest of this paper. For ease of presentation, we gather all compilation processes pertaining to control-flow based pointcuts in Section 4.

The outline of the paper is as follows: Section 2 highlights various Aspect-oriented features through AspectFun and defines its semantics. In Section 3, we describe our type inference system and the corresponding type-directed static weaving process. Next, we formulate the correctness of static weaving with respect to the semantics of AspectFun. In section 4, we provide a detailed description of how control-flow based pointcuts are handled in our compilation model. We discuss related work in Section 5, before concluding in Section 6.

## 2 AspectFun: The Aspect Language

We introduce an aspect-oriented lazy functional language, AspectFun, for our investigation. Figure 3 presents the language syntax. We write $\bar{o}$ as an abbreviation

[2] Though not presented in this paper, we have devised a deterministic type-inference algorithm to determine the well-typedness of aspect-oriented programs.

[3] The prototype is available upon request.

for a sequence of objects $o_1, ..., o_n$ (e.g. declarations, variables etc) and $\text{fv}(o)$ as the free variables in $o$. We assume that $\bar{o}$ and $o$, when used together, denote unrelated objects. We write $t_1 \sim t_2$ to specify unification. We write $t \trianglerighteq t'$ iff there exists a substitution $S$ over type variables in $t$ such that $St = t'$, and we write $t \equiv t'$ iff $t \trianglerighteq t'$ and $t' \trianglerighteq t$. To simplify our presentation, complex syntax, such as `if` expressions and sequencings (`;`), are omitted even though they are used in examples.

| | | |
|---|---|---|
| Programs | $\pi$ | $::= d$ in $\pi \mid e$ |
| Declarations | $d$ | $::= x = e \mid f\ \overline{x} = e \mid n$@advice around $\{\overline{pc}\}\ (arg) = e$ |
| Arguments | $arg$ | $::= x \mid x :: t$ |
| Pointcuts | $pc$ | $::= ppc \mid pc + cf$ |
| Primitive PC's | $ppc$ | $::= f \mid n$ |
| Cflows | $cf$ | $::=$ `cflowbelow`$(f) \mid$ `cflowbelow`$(f(\_ :: t))$ |
| Expressions | $e$ | $::= c \mid x \mid$ `proceed` $\mid \lambda x.e \mid e\ e \mid$ let $x\ =\ e$ in $e$ |
| Types | $t$ | $::= Int \mid Bool \mid a \mid t \to t \mid [t]$ |
| Advice Predicates | $p$ | $::= (f : t)$ |
| Advised Types | $\rho$ | $::= p.\rho \mid t$ |
| Type Schemes | $\sigma$ | $::= \forall \bar{a}.\rho$ |

**Fig. 3.** Syntax of the AspectFun Language

In AspectFun, top-level definitions include global variable and function definitions, as well as aspects. An *aspect* is an advice declaration which includes a piece of advice and its target *pointcuts*. An *advice* is a function-like expression that executes when any of the functions designated at the pointcut are about to execute. The act of triggering an advice during a function application is called *weaving*. Pointcuts are denoted by $\{\overline{pc}\}\ (arg)$, where $pc$ stands for either a *primitive pointcut*, represented by $ppc$, or a *composite pointcut*. Pointcuts specify certain join points in the program flow for advising. Here, we focus on join points at function invocations. Thus a primitive pointcut, $ppc$, specifies a function or advice name the invocations of which, either directly or indirectly via functional arguments, will be advised.

Advice is a function-like expression that executes *before*, *after*, or *around* a pointcut. An *around* advice is executed in place of the indicated pointcut, allowing the advised pointcut to be replaced. A special keyword `proceed` may be used inside the body of an around advice. It is bound to the function that represents "the rest of the computation" at the advised pointcut. As both *before* advice and *after* advice can be simulated by *around* advice that uses `proceed`, we only need to consider *around* advice in this paper.

A sequence of pointcuts, $\{\overline{pc}\}$, indicates the union of all the sets of join points selected by the $pc_i$'s. The argument variable $arg$ is bound to the actual argument of the named function call and it may contain a type scope. Alpha renaming is applied to local declarations beforehand so as to avoid name clash.

A composite pointcut relates the triggering of advice to the program's control flow. Specifically, we can write pointcuts which identify a subset of function invocations which occur in the dynamic context of other functions. For example, the pointcut $f +$ `cflowbelow`$(g)$ selects those invocations of $f$ which are made when the function $g$ is still executing (i.e. invoked but not returned yet).[4] As an example, in the following code, there are four invocations of `fac`, and advice `n` will be triggered by all the `fac` invocations, except the first one (`fac 3`) due to the pointcut specification "`fac+cflowbelow(fac)`".

```
n@advice around {fac + cflowbelow(fac)} (arg) = println "fac";
                                                proceed arg in
fac x = if x==0 then 1 else x * fac (x-1) in fac 3
```

Similarly, a *type-scoped* control-flow based pointcut such as `(g+cflowbelow (f(_:t)))` limits the call context to those invocations of `f` with arguments of type `t`.

Composite pointcuts are handled separately in our compilation model through series of code transformation, analyses and optimizations. This is discussed in detail in Section 4.

In AspectFun, advice names can also be primitive pointcuts. As such, we allow advices to be developed to advice other advice. We refer to such advices as *second-order advices*. In contrast, the two-layered design of AspectJ like languages only allow advices to advise other advices in a very restricted way, thus a loss in expressivity [12].

The following code fragment shows a use of second-order advice to compute the total amount of a customer order and apply discount rates according to certain business rules.

*Example 2.*

```
n3@advice around {n1,n2} (arg) = let finalRate = proceed arg
                      in  if (finalRate < 0.5) then 0.5
                             else finalRate in
n1@advice around {getRate} (arg) = (getHolidayRate arg) * (proceed arg) in
n2@advice around {getRate} (arg) = (getAnnivRate arg) * (proceed arg) in
discount item = (getRate item) * (getPrice item) in
calcPrice cart = sum (map discount cart) in ...
```

In addition to the regular discount rules, ad-hoc sale discounts such as holiday-sales, anniversary sales etc., can be introduced through aspect declarations, thus achieving separation of concern. This is shown in the `n1` and `n2` declarations. Furthermore, there may be a rule stipulating the maximum discount rate that is applicable to any product item, regardless of the multiple discounts it qualifies. Such a business rule can be realized using a second-order aspect, as in `n3`. It calls `proceed` to compute the combined discount rate and ensures that the rate do not exceed 50%.

AspectFun is polymorphic and statically typed. Central to our approach is the construct of *advised types*, $\rho$ in Figure 3, inspired by the *predicated types* [14] used in Haskell's type classes. These advised types augment common type schemes (as

[4] The semantics of `cflowbelow` adheres to that provided in AspectJ. Conversion of the popularly `cflow` pointcuts to `cflowbelow` pointcuts is available in [2].

found in the Hindley-Milner type system) with *advice predicates*, $(f : t)$, which are used to capture the need of advice weaving based on type context. We shall explain them in detail in Section 3.

We end our description of the syntax of AspectFun by referring interested readers to the accompanied technical report [2] for detailed discussion of the complete features of AspectFun, which include "catch-all" pointcut any and its variants, a diversity of composite pointcuts, nested advices, as well as advices over curried functions.

**Semantics of AspectFun.** As type information is required at the triggering of advices for weaving, the semantics of AspectFun is best defined in a language that allows dynamic manipulation of types: type abstractions and type applications. Thus, we convert AspectFun into a System-F like intermediate language, FIL.

$$
\begin{array}{lll}
\text{Program} & \pi^I & ::= (\overline{\text{Adv}}, e^I) \\
\text{Advice} & \text{Adv} & ::= (n : \varsigma, \overline{pc}, \tau, e^I) \\
\text{Join points} & jp & ::= f : \tau \mid \epsilon \\
\text{Expressions} & e^I & ::= v^I \mid x \mid \mathit{proceed} \mid e^I\ e^I \mid e^I\{\tau\} \mid \mathcal{LET}\ \ x\ =\ e^I\ \ \mathcal{IN}\ \ e^I \\
\text{Values} & v^I & ::= c \mid \lambda^{jp} x : \tau_x .\ e^I \mid \Lambda\alpha.\ e^I \\
\text{Types} & \tau & ::= \mathit{Int} \mid \mathit{Bool} \mid \alpha \mid \tau \to \tau \mid [\tau] \\
\text{Type schemes} & \varsigma & ::= \forall\overline{\alpha}.\ \tau \mid \tau
\end{array}
$$

**Fig. 4.** Syntax of FIL

$$
(\overset{\text{PROG}}{\rightarrowtail})\ \frac{\emptyset \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A}}{\pi \overset{\text{prog}}{\rightarrowtail} (\mathcal{A}, e^I)}
\qquad
(\text{Decl:MainExpr})\ \frac{\Delta \vdash e : \tau \rightarrowtail e^I}{\Delta \vdash_D e : \tau \rightarrowtail e^I; \emptyset}
$$

$$
(\text{Decl:Func})\ \frac{\begin{array}{c}\Delta.x : \tau_x \vdash e : \tau_f \rightarrowtail e^I_f \qquad \overline{\alpha} = \text{fv}(\tau_x \to \tau_f) \setminus \text{fv}(\Delta) \\ \Delta.f : \forall\overline{\alpha}.\ \tau_x \to \tau_f \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A}\end{array}}{\Delta \vdash_D f\ x{=}e \ \mathsf{in}\ \pi : \tau \rightarrowtail \mathcal{LET}\ \ f{=}\Lambda\overline{\alpha}.\ \lambda^{f:\tau_x \to \tau_f} x : \tau_x .\ e^I_f\ \ \mathcal{IN}\ \ e^I; \mathcal{A}}
$$

$$
(\text{Decl:Adv-An})\ \frac{\begin{array}{c}\text{fv}(t_x) : \text{fresh}(\text{fv}(t_x)) \vdash t_x \overset{\text{type}}{\rightarrowtail} \tau_x \\ \Delta.x : \tau_x.\mathit{proceed} : \tau_x \to \tau_n \vdash e : \tau_n \rightarrowtail e^I_n \\ \overline{\alpha} = \text{fv}(\tau_x \to \tau_n) \setminus \text{fv}(\Delta) \qquad \Delta \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A}\end{array}}{\begin{array}{r}\Delta \vdash_D n@\mathsf{advice\ around}\ \{\overline{pc}\}\ (x :: t_x) = e\ \mathsf{in}\ \pi : \tau \rightarrowtail e^I; \\ \mathcal{A}.(n : \forall\overline{\alpha}.\tau_x \to \tau_n, \overline{pc}, \tau_x, \Lambda\overline{\alpha}.\ \lambda^{n:\tau_x \to \tau_n} x : \tau_x .\ e^I_n)\end{array}}
$$

$$
(\text{Expr:Var})\ \frac{\tau = \Delta(x)}{\Delta \vdash x : \tau \rightarrowtail x}
\ (\text{Expr:Ty-App})\ \frac{\forall\overline{\alpha}.\ \tau = \Delta(x) \quad \tau_x = [\overline{\tau'}/\overline{\alpha}]\tau}{\Delta \vdash x : \tau_x \rightarrowtail x\{\overline{\tau'}\}}
$$

$$
(\text{Type:Base})\ \sigma \vdash \mathit{Int} \overset{\text{type}}{\rightarrowtail} \mathit{Int} \qquad \sigma \vdash \mathit{Bool} \overset{\text{type}}{\rightarrowtail} \mathit{Bool} \qquad \sigma.a : \alpha \vdash a \overset{\text{type}}{\rightarrowtail} \alpha
$$

$$
(\text{Type:Inferred})\ \frac{\sigma \vdash t \overset{\text{type}}{\rightarrowtail} \tau}{\sigma \vdash [t] \overset{\text{type}}{\rightarrowtail} [\tau]}
\qquad
\frac{\sigma \vdash t_1 \overset{\text{type}}{\rightarrowtail} \tau_1 \quad \sigma \vdash t_2 \overset{\text{type}}{\rightarrowtail} \tau_2}{\sigma \vdash t_1 \to t_2 \overset{\text{type}}{\rightarrowtail} \tau_1 \to \tau_2}
$$

**Fig. 5.** Conversion Rules to FIL (interesting cases)

**Expressions:**

$$(\text{OS:}\textsc{Value}) \quad c \Downarrow c \quad \lambda^{jp}x:\tau_x.\ e^I \Downarrow \lambda^{jp}x:\tau_x.\ e^I \quad \Lambda\alpha.\ e^I \Downarrow \Lambda\alpha.\ e^I$$

$$(\text{OS:}\textsc{App})\ \frac{e_1^I \Downarrow \lambda^{jp}x:\tau_x.\ e_3^I \quad \text{Trigger}(\lambda x:\tau_x.\ e_3^I, jp) = \lambda x:\tau_x.\ e_4^I \quad [e_2^I/x]e_4^I \Downarrow v^I}{e_1^I\ e_2^I \Downarrow v^I}$$

$$(\text{OS:}\textsc{Ty-App})\ \frac{e_1^I \Downarrow \Lambda\alpha.\ e_2^I \quad [\tau/\alpha]e_2^I \Downarrow v^I}{e_1^I\{\tau\} \Downarrow v^I} \qquad (\text{OS:}\textsc{Let})\ \frac{[e_1^I/x]e_2^I \Downarrow v^I}{\mathcal{LET}\ \ x\ =\ e_1^I\ \ \mathcal{IN}\ \ e_2^I \Downarrow v^I}$$

**Auxiliary Functions:**

$$\begin{array}{ll}
\text{Trigger} & :\ e^I \times jp \to e^I \\
\text{Trigger}(e^I, \epsilon) & = e^I \\
\text{Trigger}(\lambda x:\tau_x.\ e^I, f:\tau_f) & = \text{Weave}(\lambda x:\tau_x.\ e^I, \tau_f, \text{Choose}(f, \tau_x))
\end{array}$$

$$\begin{array}{ll}
\text{Weave} & :\ e^I \times \tau \times \overline{\text{Adv}} \to e^I \\
\text{Weave}(e^I, \tau_f, []) & = e^I \\
\text{Weave}(e_f^I, \tau_f, a:advs) & = \text{Let } (n:\forall\overline{\alpha}.\ \tau_n, \overline{pc}, \tau, \Lambda\overline{\alpha}.\ e^I) = a \\
 & \quad\ \text{In} \quad \text{If } \neg(\tau_n \trianglerighteq \tau_f) \text{ Then Weave}(e_f^I, \tau_f, advs) \\
 & \qquad\qquad \text{Else Let } \overline{\tau} \text{ be types such that } [\overline{\tau}/\overline{\alpha}]\tau_n = \tau_f \\
 & \qquad\qquad\qquad\qquad (e_p^I,\ e_a^I) = (\text{Weave}(e_f^I, \tau_f, advs),\ (\Lambda\overline{\alpha}.\ e^I)\{\overline{\tau}\}) \\
 & \qquad\qquad\qquad\qquad \lambda^{n:\tau_n}x:\tau_x.\ e_n^I = [e_p^I/proceed]e_a^I \\
 & \qquad\qquad\qquad \text{In} \quad \text{Trigger}(\lambda x:\tau_x.\ e_n^I, n:\tau_n)
\end{array}$$

$$\begin{array}{l}
\text{Choose}(f, \tau) = \{(n_i:\varsigma_i, \overline{pc_i}, \tau_i, e_i^I) \mid (n_i:\varsigma_i, \overline{pc_i}, \tau_i, e_i^I) \in \mathcal{A},\ \tau_i \trianglerighteq \tau, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \exists pc \in \overline{pc_i}\ s.t.\ \text{JPMatch}(f, pc)\} \\
\text{JPMatch}(f, pc) = (f \equiv pc)
\end{array}$$

**Fig. 6.** Operational Semantics for FIL

FIL stores all the advices in a separated space leaving only function declarations and the main expression in the program. Expressions in FIL, denoted by $e^I$, are extensions of those in AspectFun to include annotated lambda ($\lambda^{jp}x:\tau_x.e^I$), type abstraction ($\Lambda\alpha.e^I$) and type application ($e^I\{\tau\}$) as listed in figure 4.

The conversion is led by rule $\pi \overset{\text{prog}}{\rightarrowtail} (\mathcal{A}, e^I)$. A type environment, also called conversion environment, $\Delta$ of the structure $\overline{x:\varsigma}$ is employed. We write the judgement $\Delta \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A}$ to mean that an AspectFun program having type $\tau$ is converted to a FIL program, yielding an advice store $\mathcal{A} \in \overline{Adv}$. The judgement $\Delta \vdash e : \tau \rightarrowtail e^I$ asserts that an AspectFun *expression* $e$ having a type $\tau$ under $\Delta$ is converted to a FIL expression $e^I$. The nontrivial conversion rules are listed in Figure 5. The full set of rules is available in [2].

Specifically, the rules (Decl:Func) and (Decl:Adv-An) convert top-level function and advice declarations to ones having annotated lambda $\lambda^{f:\tau}x:\tau_x.e^I$; the annotation $\lambda^{(f:\tau)}$ highlights its jointpoint. The semantics of FIL uses these annotations to find the set of advices to be triggered. The conversion also introduces type abstraction $\Lambda\overline{\alpha}$ into the definition bodies. Rule (Expr:Ty-App) instantiates type variables to concrete types.

Each advice in AspectFun is converted to a tuple in $\mathcal{A}$. The tuple contains the advice's name ($n$) with the advice's type ($\varsigma$), the pointcuts the advice selects ($\overline{pc}$), the type-scope constraint on argument ($\tau$), and the advice body ($e^I$).

**Operational Semantics for FIL.** We describe the operational semantics for AspectFun in terms of that for FIL. Due to space limitation, we leave the semantics for handling cflow-based pointcut to [2].

The reduction-based big-step operational semantics, written as $\Downarrow_{\mathcal{A}}$, is defined in Figure 6. Together with it are definitions of the auxiliary functions used. Note that the advice store $\mathcal{A}$ is implicitly carried by all the rules, and it is omitted to avoid cluttering of symbols.

Triggering and weaving of advices are performed during function applications, as shown in rule (OS:App). Triggering operation first chooses eligible advices based on argument type, and weaves them into the function invocation – through a series of substitutions of advice bodies – for execution. Note that only those advices the types of which are instantiable to the applied function's type are selected for chaining via the Weave function.

## 3 Static Weaving

In our compilation model, aspects are woven statically (Step 5 in Figure 2). Specifically, we present in this section a type inference system which guarantees type safety and, at the same time, weaves the aspects through a type-directed translation. Note that, for composite pointcuts such as `f+cflowbelow(g)`, our static weaving system simply ignores the control-flow part and only considers the associated primitive pointcuts (ie., `f`). Treatment of control-flow based pointcuts is presented in Section 4.

**Type directed weaving.** As introduced in Section 2, *advised type* denoted as $\rho$ is used to capture function names and their types that may be required for advice resolution. We further illustrate this concept with our tracing example given in Section 1.

For instance, function `f` possesses the advised type $\forall a.(h : a \to a).a \to a$, in which $(h : a \to a)$ is called an *advice predicate*. It signifies that *the execution of any application of* `f` *may require advices of* `h` *applied with a type which should be no more general than* $a' \to a'$ *where* $a'$ *is a fresh instantiation of type variable* $a$. We say a type $t$ is more general than type $t'$ iff $t \trianglerighteq t'$ but $t \not\equiv t'$. Note that advised types are used to indicate the existence of some *indeterminate advices*. If a function contains only applications whose advices are completely determined, then the function will not be associated with an advised type; it will be associated with a normal (and possibly polymorphic) type. As an example, the type of the advised function `h` in Example 1 is $\forall a.a \to a$ since it does not contain any application of advised functions in its definition.

We begin with the following set of auxiliary functions that assists type inference:

$$\text{(Gen)} \quad gen(\Gamma, \sigma) = \forall \bar{a}.\sigma \quad \text{where } \bar{a} = \text{fv}(\sigma) \backslash \text{fv}(\Gamma) \qquad \text{(Card)} \quad |o_1 ... o_k| = k$$

The main set of type inference rules, as described in Figure 7, is an extension to the Hindley-Milner system. We introduce a judgment $\Gamma \vdash e : \sigma \rightsquigarrow e'$ to denote that expression $e$ has type $\sigma$ under type environment $\Gamma$ and it is translated to

**Expressions:**

$$(\textsc{Var})\frac{x : \forall \bar{a}.\bar{p}.t \leadsto e \in \mathit{\Gamma}}{\mathit{\Gamma} \vdash x : [\bar{t}/\bar{a}]\bar{p}.t \leadsto e} \quad (\textsc{Var-A})\frac{\begin{array}{c} x :_* \forall \bar{a}.\bar{p}.t_x \in \mathit{\Gamma} \quad t' = [\bar{t}/\bar{a}]t_x \\ wv(x : t') \quad \mathit{\Gamma} \vdash n_i : t' \leadsto e_i \\ \bar{n} : \overline{\forall \bar{b}.\bar{q}.t_n} \bowtie x \leadsto \bar{n}' \in \mathit{\Gamma} \quad \{n_i \mid t_i \unrhd t'\} \quad |\bar{y}|{=}|\bar{p}| \end{array}}{\mathit{\Gamma} \vdash x : [\bar{t}/\bar{a}]\bar{p}.t_x \leadsto \lambda \bar{y}.\langle x\ \bar{y}\ , \{e_i\}\rangle}$$

$$(\textsc{App})\ \frac{\mathit{\Gamma} \vdash e_1 : t_1 \to t_2 \leadsto e_1' \quad \mathit{\Gamma} \vdash e_2 : t_1 \leadsto e_2'}{\mathit{\Gamma} \vdash e_1\ e_2 : t_2 \leadsto (e_1'\ e_2')} \quad (\textsc{Abs})\ \frac{\mathit{\Gamma}.x : t_1 \leadsto x \vdash e : t_2 \leadsto e'}{\mathit{\Gamma} \vdash \lambda x.e : t_1 \to t_2 \leadsto \lambda x.e'}$$

$$(\textsc{Let})\ \frac{\mathit{\Gamma} \vdash e_1 : \rho \leadsto e_1' \quad \sigma = \mathit{gen}(\mathit{\Gamma}, \rho) \qquad \mathit{\Gamma}.f : \sigma \leadsto f \vdash e_2 : t \leadsto e_2'}{\mathit{\Gamma} \vdash \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : t \leadsto \mathsf{let}\ f = e_1'\ \mathsf{in}\ e_2'}$$

$$(\textsc{Pred})\ \frac{\begin{array}{c} x :_* \forall \bar{a}.\bar{p}.t_x \in \mathit{\Gamma} \quad [\bar{t}/\bar{a}]t_x \ \unrhd\ t \\ \mathit{\Gamma}.x : t \leadsto x_t \vdash e : \rho \leadsto e_t' \quad x \in A \end{array}}{\mathit{\Gamma} \vdash e : (x : t).\rho \leadsto \lambda x_t.e_t'} \ (\textsc{Rel})\ \frac{\begin{array}{c} \mathit{\Gamma} \vdash e : (x : t).\rho \leadsto e' \\ \mathit{\Gamma} \vdash x : t \leadsto e'' \quad x \in A \quad x \neq e \end{array}}{\mathit{\Gamma} \vdash e : \rho \leadsto e'\ e''}$$

**Declarations:**

$$(\textsc{Global})\ \frac{\mathit{\Gamma} \vdash e : \rho \leadsto e' \quad \sigma = \mathit{gen}(\mathit{\Gamma}, \rho) \quad \mathit{\Gamma}.id :_{(*)} \sigma \leadsto id \vdash \pi : t \leadsto \pi'}{\mathit{\Gamma} \vdash \ id = e\ \mathsf{in}\ \pi : t \leadsto \ id = e'\ \mathsf{in}\ \pi'}$$

$$(\textsc{Adv})\ \frac{\begin{array}{c} \mathit{\Gamma}.proceed : t_1 \to t_2 \vdash \lambda x.e_a : \bar{p}.t_1 \to t_2 \leadsto e_a' \quad f_i : \forall \bar{a}.t_i \in \mathit{\Gamma}_{base} \\ try(S = t_1 \ \unrhd\ t_x) \quad S(t_1 \to t_2) \ \unrhd\ t_i \\ \mathit{\Gamma}.n : \sigma \bowtie \bar{f} \leadsto n \vdash \pi : t' \leadsto \pi' \quad \sigma = \mathit{gen}(\mathit{\Gamma}, S(\bar{p}.t_1 \to t_2)) \end{array}}{\mathit{\Gamma} \vdash n@\mathsf{advice\ around}\ \{\bar{f}\}\ (x :: \forall \bar{b}.t_x) = e_a \mathsf{in}\ \pi : t' \leadsto n = e_a'\ \mathsf{in}\ \pi'}$$

**Fig. 7.** Typing rules

$e'$. We assume that the advice declarations are preprocessed and all the names which appear in any of the pointcuts are recorded in an initial global store $A$. Note that locally defined functions are not subject to being advised and not listed in $A$. We also assume that the base program is well typed in Hindley-Milner and the type information of all the functions are stored in $\mathit{\Gamma}_{base}$.

The typing environment $\mathit{\Gamma}$ contains not only the usual type bindings (of the form $x : \sigma \leadsto e$) but also *advice bindings* of the form $n : \sigma \bowtie \bar{x}$. This states that an advice with name $n$ of type $\sigma$ is defined on a set of functions $\bar{x}$. We may drop the $\bowtie \bar{x}$ part if found irrelevant. When the bound function name is advised (i.e. $x \in A$), we use a different binding $:_*$ to distinguish from the non-advised case so that it may appear in a predicate as in rule (Pred). We also use the notation $:_{(*)}$ to represent a binding which is either $:$ or $:_*$. When there are multiple bindings of the same variable in a typing environment, the newly added one shadows previous ones.

**Predicating and Releasing.** Before illustrating the main typing rules, we introduce a *weavable* constraint of the form $wv(f : t)$ which indicates that advice application of the $f$-call of type $t$ can be decided. It is formally defined as:

**Definition 1.** *Given a function $f$ and its type $t_2 \to t_2'$, if* $((\forall n.n :_{(*)} \forall \bar{a}.\bar{p}.t_1 \to t_1' \bowtie f) \in \Gamma \wedge t_1 \sim t_2) \Rightarrow t_1 \trianglerighteq t_2$, *then* $wv(f : t_2 \to t_2')$.

This condition basically means that under a given typing environment, a function's type is no more general than any of its advices. For instance, under the environment $\{n : \forall a.[a] \to [a] \bowtie f,\ n1 : \mathit{Int} \to \mathit{Int} \bowtie f\}$, $wv(f : b \to b)$ is false because the type is not specific enough to determine whether $n1$ and $n2$ should apply whereas $wv(f : \mathit{Bool} \to \mathit{Bool})$ is vacuously true and, in this case, no advice applies. Note that since unification and matching are defined on types instead of type schemes, quantified variables are freshly instantiated to avoid name capturing.

There are two rules for variable lookups. Rule (Var) is standard. In the case that variable $x$ is advised, rule (Var-A) will create a fresh instance $t'$ of the type scheme bound to $x$ in the environment. Then we check weavable condition of $(x : t')$. If the check succeeds (*i.e.*, $x$'s input type is more general or equivalent to any of the advice's), $x$ will be chained with the translated forms of all those advices defined on it, having equivalent or more general types than $x$ has (the selection is done by $\{n_i | t_i \trianglerighteq t'\}$). All these selected advices have corresponding non-advised types guaranteed by the weavable condition. This ensures the bodies of the selected advices are correctly woven. Finally, the translated expression is *normalized* by bringing all the advice abstractions of $x$ outside the chain $\langle \ldots \rangle$. This ensures type compatibility between the advised call and its advices.

If the weavable condition check fails, there must exist some advices for $x$ with more specific types, and rule (Var-A) fails to apply. Since $x \in A$ still holds, rule (Pred) can be applied, which adds an advice predicate to a type. (Note that we only allow sensible choices of $t$ constrained by $t_x \trianglerighteq t$.) Correspondingly, its translation yields a lambda abstraction with an *advice parameter.* This advice parameter enables concrete *advice-chained functions* to be passed in at a later stage, called *releasing*, through application of rule (Rel). Specifically, rule (Rel) is applied to release (*i.e.*, remove) an advice predicate from a type. Its translation generates a function application with an advised expression as argument.

**Handling Advices.** Declarations define top-level bindings including advices. We use a judgement $\Gamma \vdash \pi : \sigma \rightsquigarrow \pi'$ which reassembles the one for expressions.

Rule (Global) is very similar to rule (Let) with the tiny difference that rule (Global) binds *id* which is not in $A$ with :. It binds *id* with $:_*$ otherwise.

Rule (Adv) deals with advice declarations. We only consider type-scoped advices, and treat non-type-scoped ones as special cases having the most general type scope $\forall a.a$. We first infer a (possibly advised) type of the advice as a function $\lambda x.e_a$ under the type environment extended with `proceed`. The advice body is therefore translated. Note that this translation does not necessarily complete all the chaining because the weavable condition may not hold. Thus, as with functions, the advice is parameterized, and an advised type is assigned to it and only released when it is chained in rule (Var-A).

Next, we check whether the inferred input type is more general than the type-scope: If so, the inferred type is specialized with the substitution $S$ resulted from the matching; otherwise, the type-scope is simply ignored. The function *try* acts as an exception handler. It attempts to match two types: If the matching succeeds, a resulting substitution is assigned to $S$; otherwise, an empty substitution is returned. As a result, the inferred type $t_1$ is not strictly required to subsume the type scope $t_x$. On the other hand, the advice's type $S(t_1 \to t_2)$ is require to be more general than or equivalent to all functions' in the pointcut. Note that the type information of all the functions is stored in $\Gamma_{base}$. Finally, this advice is added to the environment. It does not appear in the translated program, however, as it is translated into a function awaiting for participation in advice chaining.

**Correctness of Static Weaving.** The correctness of static weaving is proven by relating it to the operational semantics of AspectFun. Due to space limitation, we refer readers to [2] for details.

**Example.** We illustrate the application of rules in Figure 7 by deriving the type and the woven code for the program shown in Example 1. We use $C$ as an abbreviation for $Char$. During the derivation of the definition of $f$, we have:

$$\begin{aligned}\Gamma = \{\, & h :_* \forall a.a \to a \leadsto h, n_3 : \forall a.a \to a \bowtie h \leadsto n_3,\\ & n_4 : \forall a.[a] \to [a] \bowtie h \leadsto n_4, n_5 : \forall b.[C] \to [C] \bowtie h \leadsto n_5\}\end{aligned}$$

$$\text{(P{\scriptsize RED})}\ \frac{\text{(A{\scriptsize BS})}\ \dfrac{\text{(A{\scriptsize PP})}\ \dfrac{\text{(V{\scriptsize AR})}\ \dfrac{h : t \to t \leadsto dh \in \Gamma_2}{\Gamma_2 \vdash\ h : t \to t \leadsto dh} \qquad \text{(V{\scriptsize AR})}\ \dfrac{x : t \leadsto x \in \Gamma_2}{\Gamma_2 \vdash\ x : t \leadsto x}}{\Gamma_2 = \Gamma_1, x : t \leadsto x \vdash\ (h\ x) : t \leadsto (dh\ x)}}{\Gamma_1 = \Gamma, h : t \to t \leadsto dh \vdash\ \lambda x.(h\ x) : t \to t \leadsto \lambda x.(dh\ x)}}{\Gamma \vdash\ \lambda x.(h\ x) : (h : t \to t).t \to t \leadsto \lambda dh.\lambda x.(dh\ x)}$$

Next, for the derivation of the first element of the main expression, we have:

$$\begin{aligned}\Gamma_3 = \{\, & h :_* \forall a.a \to a \leadsto h, n_3 : \forall a.a \to a \bowtie h \leadsto n_3,\ n_4 : \forall a.[a] \to [a] \bowtie h \leadsto n_4,\\ & n_5 : \forall b.[C] \to [C] \bowtie h \leadsto n_5,\ f : \forall a.(h : a \to a).a \to a \leadsto f\}\end{aligned}$$

$$\text{(A{\scriptsize PP})}\ \frac{\text{(R{\scriptsize EL})}\ \dfrac{\text{(V{\scriptsize AR})}\ \dfrac{f : \forall a.(h : a \to a).a \to a \leadsto f \in \Gamma_3}{\Gamma_3 \vdash\ f : (h : [C] \to [C]).[C] \to [C] \leadsto f} \qquad \text{(V{\scriptsize AR}-A)}\ \dfrac{h :_* \forall a.a \to a \leadsto h \in \Gamma_3 \quad \ldots}{\Gamma_3 \vdash\ h : [C] \to [C] \leadsto \langle h\ , \{n_3, n_4, n_5\}\rangle}}{\Gamma_3 \vdash\ f : [C] \to [C] \leadsto (f\ \langle h\ , \{n_3, n_4, n_5\}\rangle)} \quad \ldots}{\Gamma_3 \vdash\ (f\ \text{“c”}) : [Char] \leadsto (f\ \langle h\ , \{n_3, n_4, n_5\}\rangle\ \text{“c”})}$$

We note that rules (ABS),(LET) and (APP) are rather standard. Rule (LET) only binds $f$ with : which signalizes locally defined functions are not subject to advising.

**Final Translation and Chain Expansions.** The last step of AspectFun compilation is to expand meta-constructs produced after static weaving, such as chain-expressions, to standard expressions in AspectFun, which are called *expanded expressions*. It is in fact seperated into two steps: *addProceed* and *chain expansion*. AddProceed turns the keyword proceed into a parameter of all advices. Expansion of meta-construct (chains) is defined (partly) below by an expansion operator $[\![\cdot]\!]$. It is applied compositionally on expressions, with the help of an auxiliary function ProceedApply to substitute proper function as the proceed parameter. Moreover, ProceedApply also handles expansion of second-order advices.

$$
\begin{array}{ll}
e_M & : \text{ Expressions containing meta-constructs} \\
\text{addProceed} & : \ e_M \longrightarrow e_M \\
\text{addProceed}(\mathsf{let}\ n\ \mathit{df}\ \mathit{arg}\ =\ e_1\ \mathsf{in}\ e_2) & = \mathit{if}\ (n \text{ is an advice})\ \mathit{then} \\
 & \qquad \mathsf{let}\ n\ \mathit{df}\ \mathit{proceed}\ \mathit{arg}\ =\ e_1 \\
 & \qquad \mathsf{in}\ \text{addProceed}(e_2) \\
 & \quad \mathit{else}\ \mathsf{let}\ n\ \mathit{df}\ \mathit{arg}\ =\ e_1\ \mathsf{in}\ \text{addProceed}(e_2) \\
\text{addProceed}(e) & = e
\end{array}
$$

$$
\begin{array}{lll}
\llbracket \cdot \rrbracket & : \ e_M \longrightarrow \text{Expanded expression} & \\
\llbracket e_1\ e_2 \rrbracket & = \llbracket e_1 \rrbracket\ \llbracket e_2 \rrbracket & \textit{... trivial rules omitted} \\
\llbracket \langle f\ \overline{e}\ , \{\} \rangle \rrbracket & = \llbracket f\ \overline{e} \rrbracket & \\
\llbracket \langle f\ \overline{e}\ , \{ e_a, \overline{e_{advs}} \} \rangle \rrbracket & = \text{ProceedApply}(e_a, \langle f\ \overline{e}\ , \{ \overline{e_{advs}} \} \rangle) &
\end{array}
$$

$$
\begin{array}{lll}
\text{ProceedApply}(n\ \overline{e}, k) & = \llbracket n\ \overline{e}\ k \rrbracket & \text{if } \text{rank}(n) = 0 \\
\text{ProceedApply}(\langle n\ \overline{e}\ , \{ \overline{ns} \} \rangle, k) & = \llbracket \langle n\ \overline{e}\ k\ , \{ \overline{ns} \} \rangle \rrbracket & \text{if } \text{rank}(n) > 0
\end{array}
$$

$$
\text{rank}(x) = \begin{cases} 1 + \max_i \text{rank}(e_{a\,i}) & \text{if } x \equiv \langle f\ \overline{e}\ , \{ \overline{e_a} \} \rangle \\ 0 & \text{otherwise} \end{cases}
$$

## 4 Compiling Control-Flow Based Pointcuts

In this section, we present our compilation model for composite pointcuts – control-flow based pointcuts. Despite the fact that control-flow information are only available fully during run-time, we strive to discover as much information as possible during compilation. Our strategy is as follows: In the early stage of the compilation process (step 2 in Figure 2), we convert all control-flow based pointcuts in the source to pointcuts involving only `cflowbelow`[2]. For example,

```
m@advice around {h+cflowbelow(d(_::Int))} (arg) = ...
```

will be translated, via introduction of second-order advice, into the following:

```
m'@advice around {d} (arg :: Int) = proceed arg in
m@advice around {h+cflowbelow(m')} (arg) = ...
```

Next, the advice `m` will be further translated to

```
m@advice around {h} (arg) = ...
```

while the association of `h+cflowbelow(m')` and `m` will be remembered for future use.

After the static weaving and *addProceed* step, we reinstall the control-flow based pointcuts in the woven code through guard insertion and monad transformation (steps 6 and 8 in Figure 2), following the semantics of control-flow based pointcuts, and then subject the woven code to control-flow pointcut analysis and code optimization. The description of these steps will be presented after explaining the extension made to the FIL semantics.

**Semantics of control-flow based pointcuts.** The semantics of control-flow based pointcuts is defined by modifying the operational semantics for FIL introduced in section 2.

Specifically, we modify the operational semantics function $\Downarrow_{\mathcal{A}}$, defined in Figure 6, to carry a *stack* $\mathcal{S}$, written as $\Downarrow_{\mathcal{A}}^{\mathcal{S}}$, denoting that the progress is done under a stack environment $\mathcal{S}$. $\mathcal{S}$ is a stack of function names capturing the stack of nested calls that have been invoked but not returned at the point of reduction.

By replacing $\Downarrow$ by $\Downarrow^{\mathcal{S}}$, most of the rules remain unchanged except rules (OS:App) and (OS:Let), which are refined with the introduction of $\llparenthesis e, S \rrparenthesis$:

$$\text{(OS:App')}\ \frac{e_1^I \Downarrow^{\mathcal{S}} \lambda^{f:\tau_f} x : \tau_x .\ e_3^I \qquad \text{Trigger}'(\lambda^f x : \tau_x .\ e_3^I, f : \tau_f, \mathcal{S}) = \lambda^g x : \tau_x .\ e_4^I \\ \mathcal{S}' = \mathit{cons}(g, \mathcal{S}) \qquad [\llparenthesis e_2^I, \mathcal{S} \rrparenthesis / x] e_4^I \Downarrow^{\mathcal{S}'} v^I}{e_1^I\ e_2^I \Downarrow_{\mathcal{A}}^{\mathcal{S}} v^I}$$

$$\text{(OS:Let')}\ \frac{[\llparenthesis e_1^I, \mathcal{S} \rrparenthesis / x] e_2^I \Downarrow^{\mathcal{S}} v^I}{\mathcal{LET}\ \ x\ =\ e_1^I\ \ \mathcal{IN}\ \ e_2^I \Downarrow^{\mathcal{S}} v^I} \qquad \text{(OS:Clos)}\ \frac{e^I \Downarrow^{\mathcal{S}} v^I}{\llparenthesis e^I, \mathcal{S} \rrparenthesis \Downarrow^{\mathcal{S}'} v^I}$$

$\llparenthesis e, \mathcal{S} \rrparenthesis$ is a *stack closure*, meaning that $e$ should be evaluated under stack $\mathcal{S}$ ignoring current stack, since we adopt lazy semantics for AspectFun. Detailed discussion of the modification can be found in [2].

**State-based implementation.** As stated above, the only control-flow based pointcut to implement is the `cflowbelow` pointcut. We use an example to illustrate our implementation scheme. The following is part of a woven code after static weaving.

*Example 3.*

```
// meta-data: IFAdvice [k+cflowbelow(g)] (n,...)
n proceed arg = arg+123 in
k x = x + 1 in
g x = <k, {n}> x in
f x = if x == 0 then g x else <k, {n}> x in (f 0, f 1)
```

This first (comment) line in the code above indicates that advice `n` is associated with the pointcut `k+cflowbelow(g)`. Hence, `n` should be triggered at a call to `k` *only if* the `k`-call is made in the context of a `g`'s invocation. We call `g` the `cflowbelow` *advised function.*

In order to support the dynamic nature of the `cflowbelow` pointcut efficiently, our implementation maintains a *global state* of function invocations, and inserts state-update and state-lookup operations at proper places in the woven code. Specifically, the insertion is done at two kinds of locations: At the definitions of `cflowbelow` advised functions, `g` here, and at the uses of `cflowbelow` advices.

For a `cflowbelow` advised function definition, we encode the updating of the global state – to record the entry into and the exit from the function – in the function body. In the spirit of pure functional language, we implement this encoding using a *reader monad* [7]. In pseudo-code format, the encoding of `g` in Example 3 will be as follows:[5]

```
g x = enter "g"; <k, n> x; restore_state
```

[5] Further mechanism is required when the `cflowbelow` advised function is a built-in function. The detail is omitted here.

Here, `enter "g"` adds an entry record into the global state, and `restore_state` erases it.

Next, for each use occurrence of `cflowbelow` advices, we wrap it with a state-lookup to determine the presence of the respective pointcuts. The wrapped code is a form of *guarded expression* denoted by `<|`*guard*,`n|>` for advice `n`. It implies that `n` will be executed *only if* the *guard* evaluates to `True`. The Example 3 with wrapped code appears as follows:

*Example 3a*

```
// meta-data: IFAdvice [k+cflowbelow(g)] (n,...)
n proceed arg = arg+123 in
k x = x + 1  in
g x = enter "g"; <k, { <| isIn "g", n|> } > x; restore_state  in
f x = if x == 0 then g x else <k, { <| isIn "g", n |> } > x in (f 0, f 1)
```

The guard (`isIn "g"`) determines if `g` has been invoked and not yet returned. If so, advice `n` is executed. In this case, `n` is not triggered when evaluating `f 1`, but it is when evaluating `f 0`.

**Control-Flow Pointcut Analysis and Optimization.** From Example 3a, we note that the guard occurring in the definition of `g` is always true, and can thus be eliminated. Similarly, the guard occurring in the definition of f is always false, and the associated advice `n` can be removed from the code. Indeed, many of such guards can be eliminated during compile time, thus speeding up the execution of the woven code. We thus employ two interprocedural analyses to determine the opportunity for optimizing guarded expressions. They are **mayCflow** and **mustCflow** analyses (cf. [1]).

Since the subject language is polymorphically typed and higher-order, we adopt *annotated-type and effect* systems for our analysis (cf. [11]). We define a context $\varphi$ to be a set of function names. Judgments for both **mayCflow** and **mustCflow** analyses are of the form

$$\hat{\Gamma} \vdash e : \hat{\tau}_1 \xrightarrow{\varphi'} \hat{\tau}_2 \;\&\; \varphi$$

For **mayCflow** analysis (resp. **mustCflow** analysis), this means that under an annotated-type environment $\hat{\Gamma}$, an expression $e$ has an annotated type $\hat{\tau}_1 \xrightarrow{\varphi'} \hat{\tau}_2$ and a context $\varphi$ capturing the name of those functions which may be (resp. must be) invoked and not yet returned during the execution of $e$. The annotation $\varphi'$ above the arrow $\rightarrow$ is the context in which the function resulted from evaluation of $e$ will be invoked.

This type-and-effect approach has been described in detail in [11]. As our analyses follow this approach closely, we omit the detail here for space limitation, and refer readers to [2] for explanation. Applying both **mayCflow** and **mustCflow** analyses over the woven code given in Example 3a, we obtain the following contexts for the body of each of the functions:

$$\begin{array}{lll}
\varphi_k^{\texttt{may}} = \{f, g\}, & \varphi_g^{\texttt{may}} = \{f\}, & \varphi_f^{\texttt{may}} = \emptyset \\
\varphi_k^{\texttt{must}} = \emptyset, & \varphi_g^{\texttt{must}} = \{f\}, & \varphi_f^{\texttt{must}} = \emptyset
\end{array}$$

The result of these analyses will be used to eliminate guarded expressions in the woven code. The basic principles for optimization are:

Given a guarded expression $e_{gd}$ of the form `<| isIn` $f$, $e$ `|>`:

1. If the **mayCflow** analysis yields a context $\varphi^{\mathtt{may}}$ for $e_{gd}$ *st.* $f \notin \varphi^{\mathtt{may}}$, then the guard always fails, and $e_{gd}$ will be eliminated.
2. If the **mustCflow** analysis yields a context $\varphi^{\mathtt{must}}$ for $e_{gd}$ *st.* $f \in \varphi^{\mathtt{must}}$, then the guard always succeeds, and $e_{gd}$ will be replaced by the subexpression $e$.

Going back to Example 3a, we are thus able to eliminate all the guarded expressions, yielding the following woven code:

```
// meta-data: IFAdvice [k+cflowbelow(g)] (n,...)
n proceed arg = arg+123 in
k x = x + 1 in
g x = enter "g"; <k, {n}> x; restore_state  in
f x = if x == 0 then g x else <k, {}> x in (f 0, f 1)
```

The expression `<k,{}>` indicates that no advice is chained; thus `k` will be called as usual.

## 5 Related Work

AspectML [4,3] and Aspectual Caml [10] are two other endeavors to support polymorphic pointcuts and advices in a statically typed functional language. While they have introduced some expressive aspect mechanisms into the underlying functional languages, they have not successfully reconciled coherent and static weaving – two essential features of a compiler for an aspect-oriented functional language.

AspectML [4,3] advocates first-class join points and employs the `case-advice` mechanism to support type-scoped pointcuts based on runtime type analysis. It enables programmers to reify calling contexts and change advice behavior based on the context information found therein, thus achieving cflow based advising. Such dynamic mechanisms gives AspectML additional expressiveness not found in other works. However, many optimization opportunities are lost as advice application information is not present during compilation.

Aspectual Caml [10] takes a lexical approach to static weaving. Its weaver traverses type-annotated base program ASTs to insert advices at matched joint points. The types of the applied advices must be more general than those of the joint points, thus guaranteeing type safety. Unfortunately, the technique fails to support coherent weaving of polymorphic functions which are invoked indirectly. Moreover, there is no formal description of the type inference rules, static weaving algorithm, or operational semantics.

The implementation and optimization of AspectFun took inspirations from the AspectBench Compiler for AspectJ (ABC) [1]. Despite having a similar aim, the differences between object-oriented and functional paradigms do not allow most existing techniques to be shared. The concerns of *closures* and *inlining* can be more straightforwardly encoded with higher-order functions and function calls in AspectFun; whereas the complex control flow of higher-order functional languages makes the cflow analysis much more challenging. As a result, our typed cflow analysis has little resemblance with the one in ABC which was based on call graphs of an imperative language.

In [9], Masuhara et al. proposed a compilation and optimization model for aspect-oriented programs. As their approach employs partial evaluation to optimize a dynamic weaver implemented in Scheme, the amount of optimization is restricted by the ability of the partial evaluator. In contrast, our compilation model is built upon a static weaving framework; residues are only inserted when it is absolutely necessary (in case of some control-flow based pointcuts), which keeps the dynamic impact of weaving to a minimum.

## 6 Conclusion and Future Work

Static typing, static and coherent weaving are our main concerns in constructing a compilation model for functional languages with higher-order functions and parametric polymorphism. As a sequel to our previous work, this paper has made the following significant progress. Firstly, while the basic structure of our type system remains the same, the typing and translation rules have been significantly refined and extended beyond the two-layered model of functions and advices. Consequently, advices and advice bodies can also be advised. Secondly, we proved the soundness of our static weaving with respect to an operational semantics for the underlying language, AspectFun. Thirdly, we seamlessly incorporated a wide range of control-flow based pointcuts into our model and implemented some novel optimization techniques which take advantage of the static nature of our weaver. Lastly, we developed a compiler which follows our model to translate AspectFun programs into executable Haskell code.

Moving ahead, we will investigate additional optimization techniques and conduct empirical experiments of performance gain. Besides, we plan to explore ways of applying our static weaving system to other language paradigms. In particular, Java 1.5 has been extend with parametric polymorphism by the introduction of *generics*. Yet, as mentioned in [5], the type-erasure semantics of Java prohibits the use of dynamic type tests to handle type-scoped advices. We speculate our static weaving scheme could be a key to the solution of the problem.

## Acknowledgment

We would like to thank the anonymous referees for their insightful comments. This research is partially supported by the National University of Singapore under research grant "R-252-000-250-112", and by the National Science Council, Taiwan, R.O.C. under grant number "NSC 95-2221-E-004-004-MY2".

## References

1. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Optimising Aspect J. In: PLDI '05. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, pp. 117–128. ACM Press, NewYork (2005)
2. Chen, K., Weng, S.-C., Wang, M., Khoo, S.-C., Chen, C.-H.: A compilation model for AspectFun. Technical report, TR-03-07, National Chengchi University, Taiwan (March 2007) `http://www.cs.nccu.edu.tw/∼chenk/AspectFun/AspectFun-TR.pdf`
3. Dantas, D.S., Walker, D., Washburn, G., Weirich, S.: PolyAML: A polymorphic aspect-oriented functional programmming language. In: Proc. of ICFP'05. ACM Press, NewYork (2005)
4. Dantas, D.S., Walker, D., Washburn, G., Weirich, S.: AspectML: A polymorphic aspect-oriented functional programming language. ACM Transactions on Programming Languages and Systems (TOPLAS) (to appear, 2006)
5. Jagadeesan, R., Jeffrey, A., Riely, J.: Typed parametric polymorphism for aspects. Science of Computer Programming (to appear, 2006)
6. Jones, M.P.: Qualified Types: Theory and Practice. D.phil. thesis, Oxford University (September 1992)
7. M.P. Jones.: Functional programming with overloading and higher-order polymorphism. In: Advanced Functional Programming, pp. 97–136 (1995)
8. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
9. Masuhara, H., Kiczales, G., Dutchyn, C.: A compilation and optimization model for aspect-oriented programs. In: CC, pp. 46–60 (2003)
10. Masuhara, H., Tatsuzawa, H., Yonezawa, A.: Aspectual Caml: an aspect-oriented functional language. In: Proc. of ICFP'05. ACM Press, NewYork (2005)
11. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc, Secaucus, NJ, USA (1999)
12. Rajan, H., Sullivan, K.J.: Classpects: unifying aspect- and object-oriented language design. In: Inverardi, P., Jazayeri, M. (eds.) ICSE 2005. LNCS, vol. 4309, pp. 59–68. ACM Press, NewYork (2005)
13. Sereni, D., de Moor, O.: Static analysis of aspects. In: Aksit, M. (ed.) AOSD. 2nd International Conference on Aspect-Oriented Software Development, pp. 30–39. ACM Press, NewYork (2003)
14. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages, pp. 60–76. ACM, NewYork (1989)
15. Wang, M., Chen, K., Khoo, S.-C.: Type-directed weaving of aspects for higher-order functional languages. In: PEPM '06. Workshop on Partial Evaluation and Program Manipulation, ACM Press, NewYork (2006)

# Lattice Automata: A Representation for Languages on Infinite Alphabets, and Some Applications to Verification

Tristan Le Gall[1] and Bertrand Jeannet[2]

[1] INRIA Rennes
tlegall@irisa.fr
[2] INRIA Rhônes-Alpes
bertrand.jeannet@inrialpes.fr

**Abstract.** This paper proposes a new abstract domain for languages on infinite alphabets, which acts as a functor taking an abstract domain for a concrete alphabet and lift it to an abstract domain for words on this alphabet. The abstract representation is based on lattice automata, which are finite automata labeled by elements of an atomic lattice. We define a normal form, standard language operations and a widening operator for these automata. We apply this abstract lattice for the verification of symbolic communicating machines, and we discuss its usefulness for interprocedural analysis.

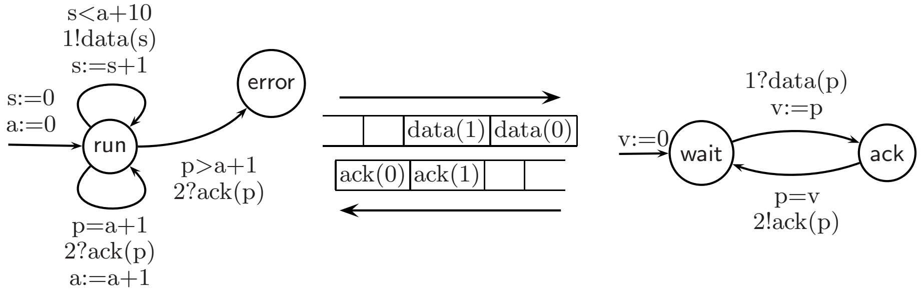
## 1 Introduction

This paper proposes a new abstract domain for languages on infinite alphabets, which acts as a functor taking an abstract domain for a concrete alphabet and lift it to an abstract domain for words on this alphabet. This abstract domain can be used not only for abstracting words, but also for abstracting the *queue* or *stack* datatypes.

The initial motivation of this work was the analysis of communicating machines exchanging messages via FIFO queues, which typically models communication protocols. In [1], we presented a reachability analysis based on abstract interpretation for protocols with a finite alphabet of messages. The set of the queue contents was abstracted by regular languages equipped with a widening operator. We want to generalize this approach to protocols with an infinite number of messages. The need for infinite alphabets comes typically from the modeling of systems sending messages with integer parameters to FIFO channels.

This is the case for the very simplified version of a sliding window protocol depicted on Fig. 1. The *sender* process tries to send data (identified by an integer) to the *receiver* process. The receiver process sends an acknowledgement messages identifying the data received. The sender has two variables : $s$ is the index of the next data to send, and $a$ is the index of the last acknowledgement message received. The protocol ensures that the sender waits for acknowledgment if $s = a + 10$. If the sender gets a message ack$(p)$ with $p > a + 1$, it means that at least one message has been lost and the protocol terminates with an error. There

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 52–68, 2007.



(a) Sender (b) Queues (c) Receiver

**Fig. 1.** A simple sliding window protocol

are two queues: one from the sender to the receiver containing data messages, the other containing acknowledgement message. Notice that we do not model here possible loss of messages.

In order to provide an abstract domain for such FIFO queues, we introduce *lattice automata*, which are finite automata in which letters belonging to a finite alphabet are replaced by elements of an atomic lattice $\Lambda$. The idea is that a language like

$$Y = \sum_{n \geq 0} data(0) \cdot \ldots \cdot data(n)$$

can be abstracted by an interval lattice automaton recognizing

$$X = data(0) + data(0) \cdot data(1) + data(0) \cdot data(1) \cdot \big(data([2, +\infty])\big)^*$$

which represents all words $data(p_0) \ldots data(p_n)$ such that $p_0 = 0$, $p_1 = 1$, and $p_k \in [2, +\infty]$ for $k \geq 2$. This idea is rather simple, but making it work properly requires a detailed study.

*Contributions.* The first contribution of this paper is to define with lattice automata an effective and canonical representation of languages on infinite alphabets, equipped with well-defined operations (union, intersection, concatenation, ...). Although the normal form we propose induces in general some approximations, it is a robust notion in the sense that the normalization operator is an upper closure operator which returns the best upper-approximation of a language in the set of normalized languages. The resulting abstract domain allows to lift any *atomic* abstract domain $A$ for $\wp(S)$ to an abstract domain $\mathrm{Reg}(A)$ for $\wp(S^*)$, the set of finite words defined on the alphabet $S$.

The second contribution of the paper is to demonstrate the use of this representation for the analysis of symbolic communicating machines (SCM). It appears indeed that this representation needs to be exploited in a clever way in order to be able to prove even simple properties, like in the example that messages in the queue 1 are always indexed by numbers smaller than the variable $s$. Our analysis shows that $a \leq s \leq a + 10$ (complete results are in section 5).

A third contribution is to show that lattice automata are also adapted to the abstraction of call-stacks in imperative programs, and allows to design potentially very precise interprocedural analysis.

*Outline.* Sect. 3 recalls some definitions about lattice and finite automata, and gives the definition of a widening operator for the regular languages lattice. The core contribution of the paper is Sect. 4 where we define lattice automata and their operations, which allows manipulating languages on infinite alphabet. In Sect. 5 we exploit this representation for the abstract interpretation of Symbolic Communicating Machines. Sect. 6 discuss the use of lattice automata for interprocedural analysis, as they allow abstracting call-stacks of imperative programs. Proofs are omitted but can be found in the companion report [2].

## 2 Related Work

Many techniques have been devoted to the analysis of Communicating Finite-State Machines (CFSM), where both the machines and the alphabet of messages are *finite*. Reachability of CFSM is undecidable [3]. Most approaches for the verification of CFSM are based on exact but semi-decidable acceleration techniques [4,5,6]. A few attack the problem with approximate techniques [7,8]. None of these works deals with a potentially infinite alphabet of messages.

More generally, there are works aiming at extending classical finite automata (resp. tree automata) representations for regular sets of words (resp. trees) for verification purposes. Mauborgne has proposed efficient representations for a class of sets of trees which strictly includes regular trees [9]. A recent work [10] introduces a different concept of "lattice automata", defined as finite (resp. Büchi) automata mapping finite (resp. infinite) words to elements of a finite lattice. Those automata do not represent finite words over an infinite alphabet, and do not apply to the verification of FIFO channels systems or interprocedural analysis. Another work [11] considers regular expressions over an infinite alphabet, which may be used instead of the lattice automata of this paper. A widening operator for regular expression is given, but does not work in the case of FIFO channels systems, because of the semantics of the send operation. Another approach is to focus on the decidability of some logic like the first order logic or the monadic second order logic when the model is a word with data or a tree with data (model of a XML document) [12]. New kind of automata were introduced, like register automata [13], pebble automata [14] or data automata [15], with the idea that a word with data satisfies the logical formula if it is recognized by the corresponding automata. This approach cannot be applied as is to our problem, as such a logical approach does not take into account any specific logical interpretation for data (in other words, the data domain is unspecified) and there is no notion of approximation.

They have been recently a lot of works devoted to shape analysis which can be relevant to the analysis of queues or stacks. A FIFO queue or a stack can indeed be represented by a list, which is most often the easiest data type to abstract in shape analysis techniques. However, most shape analysis focus on the structure ("the shape") of the memory and ignores data values hold by the memory cells [16,17,18,19]. It is sometimes possible to handle finite data values, but mainly by brute force enumeration, which is algorithmically expensive in such a context, certainly more than the above-cited approaches based on finite automata. [20] is a pioneering work for taking into account data values in memory cells. It uses a global polyhedron to relate the numeric contents of each abstract memory cells in an abstract shape graph. The resulting abstraction is incomparable to our proposal, as it is based on very different principles. In particular, the information used for abstraction in shape graphs is mostly attached to nodes instead of edges as in automata. The involved algorithms are also probably more expensive than in our solution. Conversely, it should be noted that our abstract domain could be applied to shape analysis, although this deserves yet a full study.

## 3 Preliminaries

A *finite automaton* is a quintuple $\mathcal{A} = (Q, \Sigma, Q_0, Q_f, \delta)$ where $Q$ is a finite set of states, $\Sigma$ a finite alphabet, $Q_0 \subseteq Q$ (resp. $Q_f \subseteq Q$) the set of initial (resp. final) states, and $\delta \subseteq Q \times \Sigma \times Q$ the transition relation. The set of words recognized by $\mathcal{A}$ is a regular language denoted by $L(\mathcal{A})$. $\mathrm{Reg}(\Sigma)$ is the set of regular languages over the finite alphabet $\Sigma$. Let $\approx$ be an equivalence relation on $Q$. The equivalence class of $q \in Q$ w.r.t. $\approx$ is denoted by $\widetilde{q}$. The *quotient automaton* $\mathcal{A}/\approx = \langle \widetilde{Q}, \Sigma, \widetilde{Q_0}, \widetilde{Q_f}, \widetilde{\delta} \rangle$ is defined by $\widetilde{Q} \stackrel{\triangle}{=} Q/\approx$, $\widetilde{Q_0} \stackrel{\triangle}{=} \{\widetilde{q} | q \in Q_0\}$, $\widetilde{Q_f} \stackrel{\triangle}{=} \{\widetilde{q} | q \in Q_f\}$ and $(\widetilde{q}, a, \widetilde{q'}) \in \widetilde{\delta} \stackrel{\triangle}{=} \exists q_0 \in \widetilde{q}, \exists q_0' \in \widetilde{q'} : (q_0, a, q_0') \in \delta$. Given an equivalence relation $\simeq$ on $Q$ and $k \geq 0$, the *k-depth bisimulation* equivalence relation $\approx_k$ based on $\simeq$ is defined by $\approx_0 \stackrel{\triangle}{=} \simeq$ and

$$q \approx_{k+1} q' \stackrel{\triangle}{=} \begin{cases} \forall(\sigma, q_1) \in \Sigma \times Q : \delta(q, \sigma, q_1) \Rightarrow \exists q_1' : \delta(q', \sigma, q_1') \wedge q_1 \approx_k q_1' \\ \forall(\sigma, q_1') \in \Sigma \times Q : \delta(q', \sigma, q_1') \Rightarrow \exists q_1 : \delta(q, \sigma, q_1) \wedge q_1 \approx_k q_1' \end{cases}$$

$\mathcal{A}$ is *deterministic* if there is an unique initial state and if $\delta(q, \sigma, q') \wedge \delta(q, \sigma, q'') \Rightarrow q' = q''$. Any finite automaton can be transformed into a minimal deterministic automaton (MDA) recognizing the same language, using the subset construction and quotienting the result w.r.t. the largest bisimulation relation separating final states from non final states. This provides a normal form for regular languages.

*Widening on finite automata.* $(\mathrm{Reg}(\Sigma), \subseteq)$ is a lattice which can be equipped with the following widening operator [21,1]. One first defines the extensive and idempotent operator $\rho_k$, which associates to a MDA $\mathcal{A}$ its quotient by the $k$-depth bisimulation relation based on the partition of the $Q$ which separates exactly initial, final and ordinary states. The widening $\nabla_k$ is then defined as $X_1 \nabla_k X_2 \stackrel{\triangle}{=} \rho_k(X_1 \cup X_2)$. This operator is a widening as its codomain is finite.

*Lattices.* An element $x \in \Lambda$ of a lattice $\Lambda$ is an *atom* if $\bot \sqsubset x \wedge \forall y \in \Lambda : \bot \sqsubset y \sqsubseteq x \implies y = x$. The set of atoms of $\Lambda$ is denoted by $At(\Lambda)$. A lattice $\Lambda$ is *atomic* if any element $x \in \Lambda$ is the least upper bound of atoms smaller than it: $x = \bigsqcup\{a \mid a \in At(\Lambda) \wedge a \sqsubseteq x\}$. A finite partition of a lattice is a finite set $(\lambda_i)_{0 \leq i < n}$

of elements such that $\forall i \neq j : \lambda_i \sqcap \lambda_j = \bot$ and $\forall \lambda \in \Lambda : \lambda = \bigsqcup_{i=0}^{n-1}(\lambda \sqcap \lambda_i)$. If the lattice is atomic, there is an isomorphism between an element $\lambda \in \Lambda$ and its projection $\langle \lambda \sqcap \lambda_0, \ldots, \lambda \sqcap \lambda_{n-1} \rangle$ on the partition. A (finite) *partitioning function* $\pi : \Sigma \rightarrow \Lambda$ is a function such that $(\pi(\sigma)_{\sigma \in \Sigma})$ is a (finite) partition of $\Lambda$.

## 4 Lattice Automata

In this section we define with *lattice automata* an abstract representation for languages on infinite alphabets. The principle of lattice automata is to use elements of an atomic lattice for labelling the transitions of a finite automaton, and to use a partition of this lattice in order to define a projected finite automaton which acts as a guide for defining extensions of the classical finite automata operations. We motivate our choices and show that they lead to a robust notion of approximation, in the sense that normalization is an upper-closure operation and can be seen as a best upper-approximation in the lattice of normalized lattice automata. Proofs are omitted but can be found in the companion report [2].

*Lattice automata* are finite automata, the transitions of which are labelled by elements of an atomic lattice $(\Lambda, \sqsubseteq)$ instead of elements of a finite and unstructured alphabet. They recognize languages on atomic elements of this lattice. For instance, the interval automaton of Fig. 2 recognizes all sequences of rational numbers $x_0 \ldots x_{n-1}$ where $n$ is odd, $x_{2i} \in [0, 1]$ and $x_{2i+1} \in [1, 2]$.

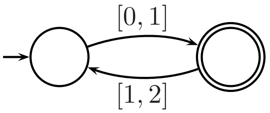


**Fig. 2.** an interval lattice automaton

**Definition 1 (Lattice automaton).** *A lattice automaton $\mathcal{A}$ is defined by a tuple $\langle \Lambda, Q, Q_0, Q_f, \delta \rangle$ where $(\Lambda, \sqsubseteq)$ is an atomic lattice, $Q$ a finite set of states, $Q_0 \subseteq Q$ and $Q_f \subseteq Q$ the sets of initial and final states and $\delta \subseteq Q \times (\Lambda \setminus \{\bot\}) \times Q$ a finite transition relation.*

A word $w = a_0 \ldots a_n \in At(\Lambda)^*$ is accepted by $\mathcal{A}$ if there exists a sequence $q_0, q_1, \ldots, q_{n+1}$ such that $q_0 \in Q_0$, $q_{n+1} \in Q_f$, and $\forall i \leq n$, $\exists (q_i, \lambda_i, q_{i+1}) \in \delta : a_i \sqsubseteq \lambda_i$. The set of words recognized by $\mathcal{A}$ is denoted by $L_{\mathcal{A}}$. We denote by $\mathrm{Reg}(\Lambda)$ the set of languages recognized by a lattice automaton. The inclusion relation between languages induces a partial order on lattice automata: $\mathcal{A} \sqsubseteq \mathcal{A}'$ iff $L_{\mathcal{A}} \subseteq L_{\mathcal{A}'}$.

*Discussion.* Def. 1 raises some issues that we discuss before refining this definition. We need indeed to provide a normal form for lattice automata, in order to use them as a well-defined abstract domain with a robust notion of approximation. The first issue is related to the bounded branching property: in a deterministic finite automaton, there are at most $|\Sigma|$ transitions outgoing from a state, whereas the branching degree of lattice automata is not bounded, *cf.* Fig. 3. The second issue is the notion of determinism. The natural definition of determinism would require that $\mathcal{A}$ is deterministic *iff* it has a unique initial state and if $(q, \lambda_1, q_1) \in \delta \wedge (q, \lambda_2, q_2) \in \delta \Longrightarrow \lambda_1 \sqcap \lambda_2 = \bot$. However the automaton

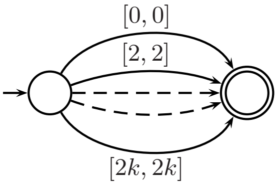
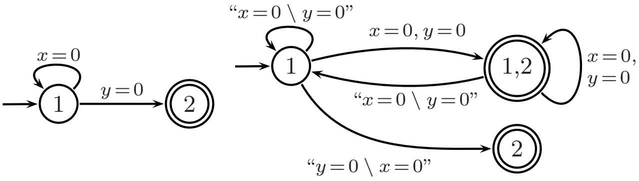


**Fig. 3.** A family of interval automata $\mathcal{A}_k$ with unbounded branching degree



(a) non-deterministic automaton

(b) deterministic automaton ?

**Fig. 4.** Attempt to determinize a lattice automaton on the lattice of affine equalities

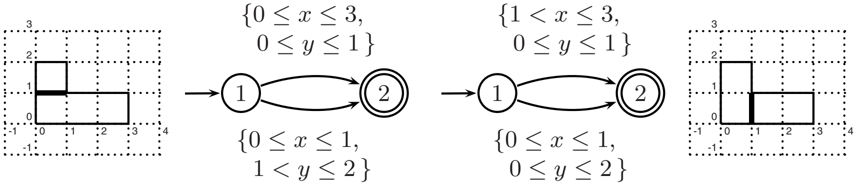


**Fig. 5.** Two deterministic convex polyhedra automata that are equivalent

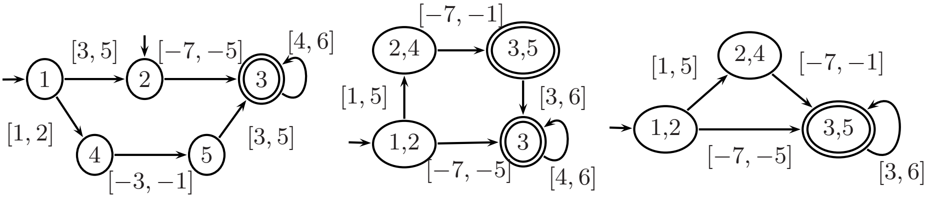
of Fig. 4 based on the lattice of affine equalities[1] shows that one cannot always find a deterministic automaton recognizing the same language as a given non-deterministic automaton, because an element of a lattice does not necessarily have a complement. Now, determinism is more or less a requirement for defining a normal form. Fig. 5 illustrates a third annoying issue : as there is no canonical representation for unions of convex polyhedra [23], how do we decide which one among the two minimal automata of the figure should be considered canonical ?

The solution we propose to fix these problems is to use a finite partition of the lattice $\Lambda$, and to decide when two transitions should be merged using the least upper bound operator. The fusion of transitions will induce in general an over-approximation, controlled by the fineness of the partition. The gain is that the projection of labels onto their equivalence class produces a finite automaton on which we can reuse classical notions.

**Definition 2 (Partitioned lattice automaton (PLA)).** *A* partitioned lattice automaton $\mathcal{A}$ *is a lattice automaton* $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$ *equipped with a finite partitioning function* $\pi : \Sigma \rightarrow \Lambda$ *such that the transition relation satisfies:* $\forall (q, \lambda, q') \in \delta,\ \exists \sigma \in \Sigma : \lambda \sqsubseteq \pi(\sigma)$.

*A PLA is* merged *if:* $(q, \lambda_1, q') \in \delta \wedge (q, \lambda_2, q') \in \delta \Longrightarrow \pi^{-1}(\lambda_1) \cap \pi^{-1}(\lambda_2) = \emptyset$

[1] The lattice of affine equalities (or affine subspaces) [22] is the lattice formed by the conjunctions of affine equalities on the space $\mathbb{R}^n$.



(a) original automaton

(b) deterministic automaton

(c) minimal automaton

**Fig. 6.** Determinization and minimization of an interval automaton with the partition $\{]-\infty, 0], [0, +\infty[\}$

**Definition 3 (Shape automaton).** *Given a PLA $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$, its shape automaton* shape($\mathcal{A}$) *is a finite automaton $(\Sigma, Q, Q_0, Q_f, \to)$ obtained by projecting the transition relation $\delta$ onto the equivalence classes:*

$$(q, \lambda, q') \in \delta \implies q \overset{\pi^{-1}(\lambda)}{\to} q'$$

Two transitions of a PLA labelled by elements belonging to different equivalence classes cannot be merged and are always kept separate, whereas they might be merged in the opposite case. Deterministic merged PLA have the finite branching degree property: its states can have at most $|\Sigma|$ outgoing transitions. Notice that non-merged PLA are as expressive as lattice automata.

**Definition 4.** *A PLA $\mathcal{A}$ is deterministic if* shape($\mathcal{A}$) *is deterministic.*

### 4.1 Normalization of PLAs

We use the shape automaton as a guideline for the determinization and minimization of lattice automata.

*Determinization.* The determinization of a PLA, which is illustrated on Fig. 6, mimics the determinization of its shape automaton using the subset construction on states. The difference is that the transitions are merged in the course of the algorithm when they are labelled by values belonging to the same equivalence class. Consequently, the resulting automaton recognizes a larger language. The precise determinization algorithm can be found in [2]. This algorithm gives the best approximation in term of inclusion of languages.

**Proposition 1 (Determinising PLA is a best upper-approximation).** *Let $\mathcal{A}$ be a PLA and $\mathcal{A}'$ the PLA obtained with the determinization algorithm sketched above. Then $\mathcal{A}'$ is the best upper-approximation of $\mathcal{A}$ as a merged and deterministic PLA, ie., $\mathcal{A} \sqsubseteq \mathcal{A}'$ and for any merged and deterministic PLA $\mathcal{A}''$ based on the same partition, $\mathcal{A} \sqsubseteq \mathcal{A}'' \implies \mathcal{A}' \sqsubseteq \mathcal{A}''$.*

**Corollary 1 (The determinization operation is an upper-closure operation).** *The operation det* : PLA → PLA *is an upper-closure operation: it is extensive ($det(\mathcal{A}) \sqsupseteq \mathcal{A}$), monotonic and idempotent ($det \circ det = det$).*

*Minimization.* We use for PLA a notion of minimization based on its shape automaton, in the same spirit as for the notion of determinism. A PLA is *minimal* or *normalized* if it is merged and if its shape automaton is minimal and deterministic. A normalized PLA will be also called a NLA (normalized lattice automaton). The algorithm to minimize a PLA consists in to remove its unconnected states, to determinize it according to the previous algorithm and to quotient it according to the equivalence bisimulation relation as defined on the states of its shape automaton (*cf.* Sect. 3). However, when quotienting the states of a PLA, transitions labelled by elements belonging to the same equivalence class are merged, which may generate some approximations, *cf.* Fig. 6.

**Theorem 1 (Minimizing PLA is a best upper-approximation).** *For any PLA $\mathcal{A}$, there is a unique (up to isomorphism) NLA $\mathcal{A}'$ based on the same partition $\pi$ such that $\mathcal{A} \sqsubseteq \mathcal{A}'$ and for any NLA $\mathcal{A}''$ based on the partition $\pi$, $\mathcal{A} \sqsubseteq \mathcal{A}'' \Longrightarrow \mathcal{A}' \sqsubseteq \mathcal{A}''$.*

**Corollary 2 (The normalization operation is an upper-closure operation).** *The normalization function* $\widehat{\cdot} : \mathrm{PLA} \to \mathrm{NLA}$ *is an upper-closure operator: it is extensive, monotonic (given a fixed partition), and idempotent.*

Thm. 1 defines a normalization for languages recognized by PLA. For any language $L \in \mathrm{Reg}(\Lambda)$ recognized by a PLA $\mathcal{A}$, $\widehat{L}$ will denote the language recognized by the unique NLA verifying the properties of Thm. 1. The set of NLA defined on $\Lambda$ with the partition $\pi$ will be denoted by $\mathrm{Reg}(\Lambda, \pi)$, which denotes also the corresponding set of recognized languages.

*Influence of the partitioning functions.* The precision of the approximations made during the merging, determinization and minimization operations depends on the fineness of the partitioning function. For example, all outgoing transitions from a given state would be merged during the determinization algorithm employed with the trivial partition of size 1.

Refining the partition $\pi$ makes the class of normalized languages $\mathrm{Reg}(\Lambda, \pi)$ more expressive. A partitioning function $\pi_2 : \Sigma_2 \to \Lambda$ *refines* $\pi_1 : \Sigma_1 \to \Lambda$ if $\forall \sigma_2 \in \Sigma_2, \exists \sigma_1 \in \Sigma_1 : \pi_2(\sigma_2) \sqsubseteq \pi_1(\sigma_1)$.

**Proposition 2.** *Let $\pi_1$ and $\pi_2$ two partitioning functions for $\Lambda$, with $\pi_2$ refining $\pi_1$. Then* $\mathrm{Reg}(\Lambda, \pi_1) \subseteq \mathrm{Reg}(\Lambda, \pi_2)$.

One can also refine PLAs before normalizing them. The automaton $\mathcal{A}_2 = \langle \Lambda, \pi_2, Q, Q_0, Q_f, \delta_2 \rangle$ *refines* $\mathcal{A}_1 = \langle \Lambda, \pi_1, Q, Q_0, Q_f, \delta_1 \rangle$ if $\pi_2$ refines $\pi_1$ and if $\delta_2$ is obtained with:

$$\frac{(q, \lambda_1, q') \in \delta_1 \qquad \sigma_2 \in \Sigma_2 \qquad \lambda_2 = \lambda_1 \sqcap \pi(\sigma_2)}{(q, \lambda_2, q') \in \delta_2}$$

This refinement does not change the recognized language, but may increase the precision of the merging, determinization and minimization algorithms:

**Proposition 3.** *Let $\mathcal{A}_1$ be a PLA and $\mathcal{A}_2$ a PLA refining $\mathcal{A}_1$. Then $det(\mathcal{A}_1) \sqsupseteq det(\mathcal{A}_2)$ and $\widehat{\mathcal{A}_1} \sqsupseteq \widehat{\mathcal{A}_2}$.*

Choosing an adequate partitioning function is thus important. For the analysis of SCM, where data messages are usually composed of a message type and some parameters, the type of the message defines a natural partition. When this standard partition is not sufficient for the analysis, it can be refined to a more adequate partition. In this sense, the abstraction refinement techniques based on partitioning (see for instance [24,25]) are applicable to lattice automata.

### 4.2 Operations on PLAs and Their Recognized Languages

*Set operations.* Two NLAs can be compared for language by first testing for inclusion their shape automata, and in case of inclusion, by comparing the labels of matching transitions. The exact union of two NLAs can be computed by considering their disjoint union; this produces a non-deterministic PLA with two initial states. Normalizing it transforms the exact union operation in an upper bound operator $\sqcup$ defined as follows: $\mathcal{A}_1 \sqcup \mathcal{A}_2 \stackrel{\triangle}{=} \widehat{\mathcal{A}_1 \cup \mathcal{A}_2}$. As a corollary of Theorem 1, this upper bound operator is actually a *least upper bound operator* on the set of NLA ordered by language inclusion. The exact intersection of two NLAs can be computed by considering their product. The result is a merged and deterministic PLA, which is not necessarily minimal. Minimizing it may induce an upper-approximation; thus, NLAs are not closed under intersection. These operations allow us to equip the set of NLA with a join semilattice structure:

**Proposition 4.** *The set of NLA defined on an atomic lattice $\Lambda$ with a fixed partition $\pi$, ordered by language inclusion, is a join semilattice: it has bottom and top elements, and a least upper bound operator. If the standard lattice operations on $\Lambda$ are computable, so are the operations on the join semilattice of NLA.*

*Other language operations.* Operation such as language concatenations or (the generalisation of) left derivation of NLAs mimics their counterparts on finite automata, with the difference that the normalization following them induces approximations in term of languages. Another useful operation for the analysis of communicating machines is the *first* operation. Left derivation and *first* operations are defined on languages as: $L/\lambda \stackrel{\triangle}{=} \{\omega \mid \exists a \in At : a \cdot \omega \in L \wedge a \sqsubseteq \lambda\}$ and *first*$(L) = \{a \in At(\Lambda) \mid a \cdot \omega \in L\}$.

*Widening on NLAs.* The widening on NLA we define here combines the widening on finite automata with the standard widening operator $\nabla_\Lambda : \Lambda \times \Lambda \rightarrow \Lambda$ that we assume for the atomic lattice $\Lambda$. If a widening operator is not strictly required for $\Lambda$ (because it satisfies the ascending chain condition), then one takes $\nabla_\Lambda = \sqcup_\Lambda$.

We first extend the $\rho_k$ operator defined on finite automata in Sect 3. If $\mathcal{A}$ is a NLA, $k \geq 0$ an integer and $\approx_k$ the k-depth bisimulation relation defined on shape($\mathcal{A}$), then $\rho_k(\mathcal{A})$ is defined as the quotient PLA $\mathcal{A}/\approx_k$. The widening operator we suggest consists in applying the operator $\rho_k$ when the two argument

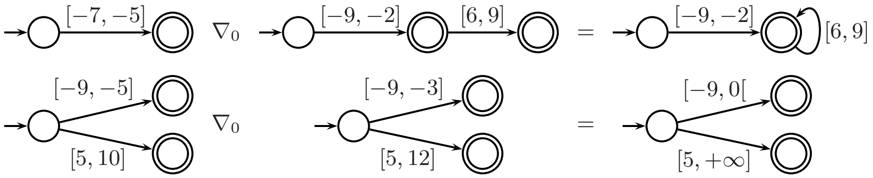


**Fig. 7.** Widening on interval PLA, with a partition $[-\infty, 0[\sqcup[0, +\infty]$ and $k = 0$

automata have a different shape automaton, and to apply the widening operator of the lattice $\Lambda$ on their matching transitions when the two argument automata have the same shape automaton, as illustrated by Fig. 7.

**Definition 5.** *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two NLA defined on the same partition $\pi$ with $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$. The widening operator $\nabla_k$ is defined as:*

$$\mathcal{A}_1 \nabla_k \mathcal{A}_2 \triangleq \begin{cases} \widehat{\rho_k(\mathcal{A}_2)} & \textit{if } \mathrm{shape}(\mathcal{A}_1) \neq \mathrm{shape}(\rho_k(\mathcal{A}_2)) \\ \mathcal{A}_1 \nearrow \mathcal{A}_2 & \textit{otherwise (which implies } \mathrm{shape}(\mathcal{A}_1) = \mathrm{shape}(\mathcal{A}_2)\textit{)} \end{cases}$$

*where $\mathcal{A}_1 \nearrow \mathcal{A}_2$ is the NLA $\mathcal{A}$ which has the same set of states as $\mathcal{A}_1$ and $\mathcal{A}_2$ and the set of transitions $\delta$ defined by the rule:*

$$\frac{\sigma \in \Sigma \quad (q, \lambda_1, q') \in \delta_1 \quad (q, \lambda_2, q') \in \delta_2 \quad \lambda_1, \lambda_2 \sqsubseteq \pi(\sigma)}{(q, (\lambda_1 \nabla_\Lambda \lambda_2) \sqcap \pi(\sigma), q') \in \delta}$$

*If $\mathcal{A}_1 \not\sqsubseteq \mathcal{A}_2$, then $\mathcal{A}_1 \nabla_k \mathcal{A}_2 \triangleq \mathcal{A}_1 \nabla_k (\mathcal{A}_1 \sqcup \mathcal{A}_2)$.*

**Theorem 2.** *$\nabla_k$ is a proper widening operator:*

1. *for any NLA $\mathcal{A}1, \mathcal{A}_2$ defined on the same partition such that $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$, $\mathcal{A}_2 \sqsubseteq \mathcal{A}_1 \nabla_k \mathcal{A}_2$;*
2. *If there is an increasing chain of NLA $\mathcal{A}_0 \sqsubseteq \mathcal{A}_1 \sqsubseteq \ldots \sqsubseteq \mathcal{A}_n \sqsubseteq \ldots$, the chain $\mathcal{A}'_0 \sqsubseteq \mathcal{A}'_1 \sqsubseteq \ldots \sqsubseteq \mathcal{A}'_n \sqsubseteq \ldots$ defined as $\mathcal{A}'_0 = \mathcal{A}_0$ and $\mathcal{A}'_{i+1} = \mathcal{A}'_i \nabla_k (\mathcal{A}'_i \sqcup \mathcal{A}_{i+1})$ is not strictly increasing.*

### 4.3 NLA as an Abstract Domain for Languages, Stacks and Queues

Normalized lattice automata allow us to define an *abstract domain functor* which lifts abstract domains for some set to abstract domains for languages on this set. More precisely, given an *atomic* abstract lattice $A \xrightarrow{\gamma_A} \wp(S)$ for some set $S$ with the concretization function $\gamma_A$, and a partitioning function $\pi$ for $A$, $\mathrm{Reg}(A, \pi)$ can be viewed as an abstract domain for $\mathcal{L}(S) = \wp(S^*)$, the languages on elements of $S$, with the concretization function

$$\begin{aligned} \gamma : \mathrm{Reg}(A, \pi) &\to \wp(S^*) \\ \mathcal{A} &\mapsto \{s_0 \ldots s_n \in S^* \mid a_0 \ldots a_n \in L_{\mathcal{A}} \wedge \forall i : s_i \in \gamma_A(a_i)\} \end{aligned}$$

and the widening operator of Def. 5. $\mathrm{Reg}(A, \pi)$ is a non-complete join semilattice of infinite height.

The "abstract domain functor" $\mathrm{Reg}(A, \pi)$ is in addition monotonic in the following sense. Given two abstractions $\gamma_i : A_i \to C, i = 1, 2$, for the same concrete domain $C$, $A_2$ *refines* $A_1$ if there exists a (concretization) function $\gamma_{12} : A_1 \to A_2$ such that $\gamma_1 = \gamma_2 \circ \gamma_{12}$. This means that any concrete property $c \in C$ representable by $A_1$ is representable by $A_2$.

**Theorem 3.** *Let $\gamma_i : A_i \to \wp(S), i = 1, 2$, two abstractions for $\wp(S)$ such that $A_2$ refines $A_1$, with $\gamma_{12} : A_1 \to A_2$ such that $\gamma_1 = \gamma_2 \circ \gamma_{12}$. Let $\pi_1$ a partitioning function for $A_1$, and $\pi_2$ a partitioning function for $A_2$ refining $\gamma_{12} \circ \pi_1$. The abstract domain* $\mathrm{Reg}(A_2, \pi_2)$ *refines* $\mathrm{Reg}(A_1, \pi_1)$.

Hence an abstraction of $\wp(S^*)$ based on lattice automata is parametrised first by the underlying alphabet lattice $A$, then by a partition $\pi$ of $A$, and last by the parameter $k$ of the widening operator $\nabla_k$.

Most language operations can be efficiently abstracted in $\mathrm{Reg}(A, \pi)$. This is in particular the case of language operations corresponding to the operations offered by the *FIFO queue* or *stack* abstract datatypes. Hence, $\mathrm{Reg}(A, \pi)$ is a suitable abstract domain for FIFO queues or stacks on elements of $S$.

## 5 Application to the Abstract Interpretation of SCMs

We illustrate in this section the application of the abstract lattice defined in the previous section to the analysis of Symbolic Communicating Machines (SCMs).

**Symbolic Communicating Machines.** Symbolic Communicating Machines (SCM) are Communicating Finite-State Machine extended with a finite set of variables $V$, the values of which can be sent into FIFO queues, *cf.* Fig. 1. This model is similar to other models like Extended Communicating Finite-State Machines [26] or Parametrized Communicating Finite-State Machines [27].

*Syntax.* A SCM with $N$ queues is defined by a tuple $\langle C, V, c_0, \Theta_0, P, \Delta \rangle$ where $C$ is a nonempty finite set of locations; $V = \{v_1, \ldots, v_n\}$ is a nonempty, finite set of *state* variables; $c_0 \in C$ is the initial control state; $\Theta_0(\boldsymbol{v})$ is the *initial condition*; $P = \{p_1, \ldots, p_n\}$ is a nonempty, finite set of formal parameters that are used to push/pop values on/from FIFO queues; and $\Delta$ is a finite set of *transitions.*

A transition $\delta$ is either an *input* $\langle c_1, G, i?\boldsymbol{p}, A, c_2 \rangle$ or an *output* $\langle c_1, G, i!\boldsymbol{p}, A, c_2 \rangle$ where $c_1$ and $c_2$ are resp. the origin and destination locations; $i \in [1..N]$ is a queue number; $\boldsymbol{p}$ is the vector of formal parameters, which holds the values pushed or popped on/from the queue $i$; $G(\boldsymbol{v}, \boldsymbol{p})$ is a predicate on the variables and the formal parameters (also called guard); and $A$ is an assignment of the form $\boldsymbol{v}' := A(\boldsymbol{v}, \boldsymbol{p})$ defining the values of the state variables after the transition.

*Semantics.* In the sequel, a variable $v$ takes its value in a set denoted by $\mathcal{D}_v$, and the set of valuations of all variables in $V$ is denoted by $D_V$. A global state of a SCM is a tuple $\langle c, \boldsymbol{v}, w_1, \ldots, w_N \rangle \in C \times \mathcal{D}_V \times ((\mathcal{D}_P)^*)^N$ where $c$ is a control state, $\boldsymbol{v}$ is the current value of the variables and $w_i$ is a finite word on $\mathcal{D}_P$ representing the content of queue $i$. The operational semantics of a SCM

$\langle C, V, \Sigma, c_0, \Theta_0, P, \Delta\rangle$ is given as an infinite transition system $\langle Q, Q_0, \rightarrow\rangle$ where $Q = C \times \mathcal{D}_V \times ((\mathcal{D}_P)^*)^N$ is the set of states; $Q_0 = \{\langle c_0, \boldsymbol{v}, \varepsilon, \ldots, \varepsilon\rangle \mid \boldsymbol{v} \in \Theta_0\}$ is the set of the initial states; and $\rightarrow$ is defined by the two rules:

$$\frac{(c_1, G, i!\boldsymbol{p}, A, c_2) \in \Delta \qquad w_i' = w_i \cdot \boldsymbol{p} \qquad G(\boldsymbol{v}, \boldsymbol{p}) \qquad \boldsymbol{v'} = A(\boldsymbol{v}, \boldsymbol{p})}{\langle c_1, \boldsymbol{v}, w_1, \ldots, w_i, \ldots, w_N\rangle \rightarrow \langle c_2, \boldsymbol{v'}, w_1, \ldots, w_i', \ldots, w_N\rangle}$$

$$\frac{(c_1, G, i?\boldsymbol{p}, A, c_2) \in \Delta \qquad w_i = \boldsymbol{p}.w_i' \qquad G(\boldsymbol{v}, \boldsymbol{p}) \qquad \boldsymbol{v'} = A(\boldsymbol{v}, \boldsymbol{p})}{\langle c_1, \boldsymbol{v}, w_1, \ldots, w_i, \ldots, w_N\rangle \rightarrow \langle c_2, \boldsymbol{v'}, w_1, \ldots, w_i', \ldots, w_N\rangle}$$

The main issue for the reachability analysis of SCM is the abstraction of the queue contents. [1] described and experimented the abstraction of the queue contents of a CFSM using the regular languages lattice equipped with the widening operator of Sect. 3. Lattice automata allow us to attack the analysis of SCM. For the sake of simplicity, we assume a single queue in the sequel. The running example which has two queues is analysed with a *non-relational, attribute-independent* method, in which to each queue is associated a lattice automaton.

**Analysing SCM with a single queue: a simple approach.** If there is a single queue, the concrete set of states associated to each control point $c \in C$ has the structure $\wp(\mathcal{D}_V \times (\mathcal{D}_P)^*)$: one associates to each control point the set of possible configurations for the state variables and the FIFO queue. For the sake of simplicity, we will assume in the sequel that all variables and parameters are of rational type, and that sets of valuations are abstracted using convex polyhedra. Let $L^{(n)} = \mathsf{Pol}(\mathbb{Q}^n)$ denotes the atomic lattice of convex polyhedra. We have the abstraction chain

$$\wp(\mathbb{Q}^n \times (\mathbb{Q}^p)^*) \leftrightarrows \wp(\mathbb{Q}^n) \times \wp((\mathbb{Q}^p)^*) \longleftarrow L^{(n)} \times \mathrm{Reg}(L^{(p)}) \overset{\triangle}{=} A^{(n)}$$

Tab. 1 gives the corresponding abstract semantics.

We experimented with this analysis on the sliding window protocol depicted in Fig. 1, using a generic fixpoint calculator and the APRON library [28], with no partitioning of the alphabet lattice. Tab. 2 depicts the obtained reachable set on the product of the two automata. This straightforward abstraction is not very accurate since there is no relation between the values of parameter variables in the queue and the value of the state variables.

**SCM with a single queue: linking message and state variables.** The idea to improve on the previous abstraction is to use an augmented semantics in which both the state variables and the message are pushed on queues. This allows not only to establish relations between messages in queues and the current environment, but also to indirectly establish relations between the messages contained in different queues. For instance, the abstract value

$$\big(s \in [8, 9],\ \ \mathit{data}(\{s - p = 3\}) \cdot \mathit{data}(\{s - p = 2\}) \cdot \mathit{data}(\{s - p = 1\})\big)$$

will represent the 2 concrete states $(s = 8, \mathit{data}(5) \cdot \mathit{data}(6) \cdot \mathit{data}(7))$ and $(s = 9, \mathit{data}(6) \cdot \mathit{data}(7) \cdot \mathit{data}(8))$.

Formally, the new abstraction is defined by

$$\begin{array}{lll} \gamma : & L^{(n)} \times \mathrm{Reg}(L^{(n+p)}) \rightarrow \wp(Q^n \times (Q^p)^*) & \\ & (Y, F) \qquad\qquad \mapsto \{(\boldsymbol{v}, (\boldsymbol{v}, \boldsymbol{p_0}) \ldots (\boldsymbol{v}, \boldsymbol{p_k})) \mid \boldsymbol{v} \in Y \wedge (\boldsymbol{v}, \boldsymbol{p_0}) \ldots (\boldsymbol{v}, \boldsymbol{p_k})) \in L_F\} & \end{array}$$

**Table 1.** Abstract semantics, with $L^{(n)} = \mathsf{Pol}(\mathbb{Q}^n)$ and $A^{(n)} = L^{(n)} \times \mathrm{Reg}(L^{(p)})$

| Standard abstract semantics | | |
|---|---|---|
| guard | $G(\boldsymbol{v},\boldsymbol{p})$ | $[\![G]\!]^\sharp \ : \ L^{(n)} \to L^{(n+p)}$ extended to $A^{(n)} \to A^{(n+p)}$ |
| assignment | $A(\boldsymbol{v},\boldsymbol{p})$ | $[\![A]\!]^\sharp \ : L^{(n+p)} \to L^{(n)}$ extended to $A^{(n+p)} \to A^{(n)}$ |
| output | $1!\boldsymbol{p}$ | $[\![1!\boldsymbol{p}]\!]^\sharp : A^{(n+p)} \to A^{(n+p)}$<br>$(Y,F) \mapsto (Y, F \cdot (\exists \boldsymbol{v} : Y))$ |
| input | $1?\boldsymbol{p}$ | $[\![1?\boldsymbol{p}]\!]^\sharp : A^{(n+p)} \to A^{(n+p)}$<br>$(Y,F) \mapsto (Y \sqcap \mathit{first}(F),\ F/(\exists \boldsymbol{v} : Y))$ |
| transition | $t$ | $[\![t]\!]^\sharp \ : \ A^{(n)} \to A^{(n)}$<br>$X \mapsto \begin{cases} [\![A]\!]^\sharp \circ [\![G]\!]^\sharp & \text{if } t = (G, --, A) \\ [\![A]\!]^\sharp \circ [\![1!\boldsymbol{p}]\!]^\sharp \circ [\![G]\!]^\sharp & \text{if } t = (G, 1!\boldsymbol{p}, A) \\ [\![A]\!]^\sharp \circ [\![1?\boldsymbol{p}]\!]^\sharp \circ [\![G]\!]^\sharp & \text{if } t = (G, 1?\boldsymbol{p}, A) \end{cases}$ |

**Table 2.** Analysis with the "simple" abstraction

| $C$ | Abstract Value |
|---|---|
| $rw$ | $[\lvert v \geq 0; s \geq 0; a \geq 0\rvert]$<br>$(d, [\lvert p \geq 0\rvert])^*$<br>$(a, [\lvert p \geq 0\rvert])^*$ |
| $ra$ | $[\lvert v \geq 0; s \geq 0; a \geq 0\rvert]$<br>$(d, [\lvert p \geq 0\rvert])^*$<br>$(a, [\lvert p \geq 0\rvert])^*$ |
| $ew$ | $[\lvert v \geq 0; s \geq 0; a \geq 2\rvert]$<br>$(d, [\lvert p \geq 0\rvert])^*$<br>$(a, [\lvert p \geq 0\rvert])^*$ |
| $ea$ | $[\lvert v \geq 0; s \geq 0; a \geq 2\rvert]$<br>$(d, [\lvert p \geq 0\rvert])^*$<br>$(a, [\lvert p \geq 0\rvert])^*$ |

**Table 3.** Analysis with the refined abstraction, using a widening "up to" for polyhedra

| $C$ | Abstract Value |
|---|---|
| $rw$ | $[\lvert 0 \leq a \leq s \leq a+10\rvert]$<br>$(d, [\lvert 0 \leq a \leq s-1 \leq a+9; 0 \leq p \leq s-1\rvert])^*$<br>$(a, [\lvert 0 \leq a \leq s \leq a+10; 0 \leq p \leq s-1; 0 \leq v \leq s-1\rvert])^*$ |
| $ra$ | $[\lvert 0 \leq a \leq s \leq a+10; 0 \leq v \leq s-1\rvert]$<br>$(d, [\lvert 0 \leq a \leq s-1 \leq a+9; 0 \leq p \leq s-1\rvert])^*$<br>$(a, [\lvert 0 \leq a \leq s \leq a+10; 0 \leq p \leq s-1; 0 \leq v \leq s-1\rvert])^*$ |
| $ew$ | $[\lvert 0 \leq a \leq s-2 \leq a+8; 0 \leq v \leq s-1\rvert]$<br>$(d, [\lvert 0 \leq a \leq s-1 \leq a+9; 0 \leq p \leq s-1\rvert])^*$<br>$(a, [\lvert 0 \leq a \leq s \leq a+10; 0 \leq p \leq s-1; 0 \leq v \leq s-1\rvert])^*$ |
| $ea$ | $[\lvert 0 \leq a \leq s-2 \leq a+8; 0 \leq v \leq s-1\rvert]$<br>$(d, [\lvert 0 \leq a \leq s-1 \leq a+10; 0 \leq p \leq s-1\rvert])^*$<br>$(a, [\lvert 0 \leq a \leq s \leq a+10; 0 \leq p \leq s-1; 0 \leq v \leq s-1\rvert])^*$ |

The main difference of the induced abstract semantics w.r.t. the previous one is that the operators $[\![G]\!]$ and $[\![A]\!]$ now modify the lattice automaton representing the queue content, because one must modify the transition labels each time the state variables are altered.

We obtained the results of Tab. 3 with this refined abstraction, using the widening $\nabla_k$ with $k = 0$. In particular, we proved that $0 \leq a \leq s \leq a + 10$, $0 \leq p \leq s-1$ and $0 \leq v \leq s$. The two control states $ew$[2] and $ea$ are however still reachable, whereas they are not in the real system. An exact analysis should give the invariant $a \leq v \leq s \leq a+10$ with first queue $p = s-1, p = s-2, \ldots, p = v+1$ and second queue $p = v, p = v-1, \ldots, p = a+1$. Those relations are lost when merging transitions of the automata representing the queues. This abstraction is far better than the first one, but should still be improved.

[2] Control states are represented by two letters $xy$, encoding the control states of the sender (r = run,e= error) and of the receiver (w=wait,a=ack).

## 6 Application to Interprocedural Analysis

We discuss here an interesting application of lattice automata to precise interprocedural analysis, based on the abstraction of call-stacks of imperative programs. This would allow simplifying the stack abstraction proposed in [8] and to extend to infinite state programs the approach of [29] which uses pushdown automata to model finite-state recursive programs and finite automata for representing (co)reachable sets of configurations.

The concrete semantics of imperative programs without global variables makes use of the domains depicted on the following table:

| | | |
|---|---|---|
| $c$ | $\in C$ | : control points |
| $\epsilon_i$ | $\in \mathit{LEnv}_i = \mathit{LVar}_i \rightarrow \mathit{Value}$ | : local environments for procedure/function $P_i$ |
| $\langle c, \epsilon \rangle$ | $\in \mathit{Act} = C \times \mathit{LEnv}$ | : activation record ($\mathit{LEnv} = \bigcup_i \mathit{LEnv}_i$) |
| $\Gamma$ | $\in \mathit{Act}^+$ | : stacks = program states |

Given an abstraction $\wp(\mathit{LEnv}) \leftrightarrows L$ for environments, we can use the abstraction:

$$\wp(\mathit{Act}^+) \xleftarrow{\gamma} \mathrm{Reg}(C \times L, \pi)$$

with the partitioning function $\pi(c) = \{c\} \times \top_L$ based on control points. Lattice automata provide all the necessary operations to abstract the concrete semantics of such imperative programs (*ie.*, push and pop operations, modifications of the top of the stack).

There are some advantages in having an explicit representation of call-stacks in interprocedural analysis. As long as data variables are not abstracted, the more classical functional approach is as precise, but as soon as they are, such an explicit representation allows recovering some loss of information. Moreover, the stack abstraction approach allows a natural description of techniques such as polyvariant analysis, where separate analyses are performed on the same procedure depending on the call context. Last, some applications *require* information on the stack. This is the case for instance of analysis related to stack inspection mechanisms for ensuring security properties in Java and .NET architecture [30]. Another example is the test selection technique proposed in [31], in the context of conformance testing of reactive systems w.r.t. an interprocedural specification.

*Interprocedural analysis by explicit representation of the call-stacks.* The call-string approach of [32], generalised by tokens in [33] corresponds to a very strong abstraction of the call-stacks in which the value of variables is largely ignored. It is more suitable to compilation-oriented dataflow analysis than to program verification. A different line of work consists in modelling recursive programs using pushdown automata and exploiting the property that the set of (co)reachable stacks of pushdown automata is regular [34] and its computation has a polynomial complexity [35,36,37]. The use of lattice automata can be seen as an extension of this line of work to more expressive, non finite-state programs, where undecidability is overcome by resorting to approximations. The lattice automata abstraction may be seen as both an improvement and a simplification of [8], which derives and unifies two classical techniques for interprocedural analysis (namely the call-string and functional approaches identified by [32]) by abstract interpretation of the operational semantics of imperative programs. In the context of interprocedural shape analysis, [38] represents explicitly the call-stack using the same abstract representation than for the memory configurations.

## 7 Conclusion

We defined in this paper an abstract domain for languages on infinite alphabets, which are represented by *lattice automata*. Their principle is to use elements of an atomic lattice for labelling the transitions of a finite automaton, and to use a partition of this lattice in order to define a projected finite automaton which acts as a guide for defining determinization, minimization and widening operations. We motivate our choices and show that they lead to a robust notion of approximation, in the sense that normalization is an upper-closure operation and can be seen as a best upper-approximation in the join semilattice of normalized lattice automata.

The resulting abstract domain allows to lift any atomic abstract domain $A$ for $\wp(S)$ to an abstract domain $\mathrm{Reg}(A)$ for $\wp(S^*)$. It is also *parametric*: it is parametrised first by the underlying alphabet lattice, then by a partition of the alphabet lattice, and last by the parameter of the widening operator. Its precision may be improved by adjusting these parameters.

We illustrate the use of lattice automata for the verification of symbolic communicating machines, and we show the need for a non-standard semantics to couple the abstraction of the state variables of the machines with the contents of the FIFO queues. We also explore the applicability of lattice automata to interprocedural analysis and compare this solution to related work. As a result, this work allows to extend both analysis techniques dedicated to communicating machines, and interprocedural analysis based on an explicit representation of the call-stacks.

Future work includes first a deeper study of the application of lattice automata to the analysis of SCM. The challenge we would like to take up is the verification of the SSCOP communication protocol, which is a sliding window protocol from which our running example is extracted. Previous verification attempts that we are aware of are either based on enumerated state-space exploration techniques [39], or on the partial use of theorem proving [40]. It would be interesting to study the application of lattice automata to shape analysis, in the spirit of [19]. A last direction which could be explored is the generalisation of lattice automata recognizing languages on infinite alphabets to tree automata recognizing trees on infinite sets of symbols.

## References

1. Le Gall, T., Jeannet, B., Jéron, T.: Verification of communication protocols using abstract interpretation of FIFO queues. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, Springer, Heidelberg (2006)

2. Le Gall, T., Jeannet, B.: Analysis of communicating infinite state machines using lattice automata. Technical Report PI 1839, IRISA (2007)
3. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. of ACM 30(2) (1983)
4. Boigelot, B., Godefroid, P., Willems, B., Wolper, P.: The power of QDDs. (extended abstract) In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, Springer, Heidelberg (1997)
5. Bouajjani, A., Habermehl, P.: Symbolic reachability analysis of FIFO-channel systems with nonregular sets of configurations. Theoretical Computer Science 221(1-2) (1999)
6. Finkel, A., Iyer, S.P., Sutre, G.: Well-abstracted transition systems: application to FIFO automata. Information and Computation 181(1) (2003)
7. Peng, W., Puroshothaman, S.: Data flow analysis of communicating finite state machines. ACM Trans. Program. Lang. Syst. 13(3) (1991)
8. Jeannet, B., Serwe, W.: Abstracting call-stacks for interprocedural verification of imperative programs. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, Springer, Heidelberg (2004)
9. Mauborgne, L.: Tree schemata and fair termination. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, Springer, Heidelberg (2000)
10. Kupferman, O., Lustig, Y.: Lattice automata. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 199–213. Springer, Heidelberg (2007)
11. Logozzo, F.: Separate compositional analysis of class-based object-oriented languages. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 332–346. Springer, Heidelberg (2004)
12. Neven, F., Schwentick, T., Vianu, V.: Towards regular languages over infinite alphabets. In: Sgall, J., Pultr, A., Kolman, P. (eds.) MFCS 2001. LNCS, vol. 2136, Springer, Heidelberg (2001)
13. Kaminski, M., Francez, N.: Finite-memory automata. Theoretical Computer Science 134(2) (1994)
14. Milo, T., Suciu, D., Vianu, V.: Typechecking for XML transformers. In: Symp. on Principles of Database Systems (2000)
15. Bojanczyk, M., Muscholl, A., Schwentick, T., Segoufin, L., David, C.: Two-variable logic on words with data. In: Symp. on Logic in Computer Science, LICS '06 (2006)
16. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Symp. on Principles of Programming Languages (POPL'99) (1999)
17. Yavuz-Kahveci, T., Bultan, T.: Automated verification of concurrent linked lists with counters. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, Springer, Heidelberg (2002)
18. Distefano, D., O'Hearn, P., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, Springer, Heidelberg (2006)
19. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract tree regular model checking of complex dynamic data structures. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, Springer, Heidelberg (2006)
20. Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M.: Numeric domains with summarized dimensions. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, Springer, Heidelberg (2004)
21. Feret, J.: Abstract interpretation-based static analysis of mobile ambients. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, Springer, Heidelberg (2001)
22. Karr, M.: Affine relationships among variables of a program. Acta Informatica 6 (1976)

23. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Symp. on Principles of programming languages, POPL '78 (1978)
24. Jeannet, B.: Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. Formal Methods in System Design 23(1) (2003)
25. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, Springer, Heidelberg (2005)
26. Higuchi, M., Shirakawa, O., Seki, H., Fujii, M., Kasami, T.: A verification procedure via invariant for extended communicating finite-state machines. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, Springer, Heidelberg (1993)
27. Lee, D., Ramakrishnan, K.K., Moh, W.M., Shankar, U.: Protocol specification using parameterized communicating extended finite state machines. In: Int. Conf. on Network Protocols, ICNP'96 (1996)
28. Jeannet, B., Miné, A.: The APRON Numerical Abstract Domain Library. `http://apron.cri.ensmp.fr/library/`
29. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, Springer, Heidelberg (2001)
30. Besson, F., Jensen, T., Métayer, D.L., Thorn, T.: Model checking security properties of control flow graphs. J. of Computer Security 9 (2001)
31. Constant, C., Jeannet, B., Jéron, T.: Automatic test generation from interprocedural specifications. Technical Report PI 1835, IRISA Submitted to TEST-COM/FATES conference (2007)
32. Sharir, M., Pnueli, A.: Semantic foundations of program analysis. In: Program Flow Analysis: Theory and Applications (1981)
33. Jones, N.D., Muchnick, S.S.: A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In: Symp. on Principles of Programming Languages (POPL'82) (1982)
34. Caucal, D.: On the regular structure of prefix rewriting. Theoretical Computer Science 106 (1992)
35. Esparza, J., Knoop, J.: An automata-theoretic approach to interprocedural data-flow analysis. In: Thomas, W. (ed.) ETAPS 1999 and FOSSACS 1999. LNCS, vol. 1578, Springer, Heidelberg (1999)
36. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, Springer, Heidelberg (1997)
37. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. Electronic Notes on Theoretical Computer Science 9 (1997)
38. Rinetzky, N., Sagiv, M.: Interprocedural shape analysis for recursive programs. In: Wilhelm, R. (ed.) CC 2001 and ETAPS 2001. LNCS, vol. 2027, Springer, Heidelberg (2001)
39. Bozga, M., Fernandez, J.C., Ghirvu, L., Jard, C., Jéron, T., Kerbrat, A., Morel, P., Mounier, L.: Verification and test generation for the SSCOP protocol. Scientific Computer Programming 36(1) (2000)
40. Rusu, V.: Combining formal verification and conformance testing for validating reactive systems. J. of Software Testing, Verification, and Reliability 13(3) (2003)

# Compositional Verification and 3-Valued Abstractions Join Forces[⋆]

Sharon Shoham and Orna Grumberg

Computer Science Department, Technion, Haifa, Israel
{sharonsh,orna}@cs.technion.ac.il

**Abstract.** Two of the most promising approaches to fighting the state explosion problem are abstraction and compositional verification. In this work we join their forces to obtain a novel fully automatic compositional technique that can determine the truth value of the full $\mu$-calculus with respect to a given system.

Given a system $M = M_1 \| M_2$, we view each component $M_i$ as an abstraction $M_i\!\uparrow$ of the global system. The abstract component $M_i\!\uparrow$ is defined using a 3-valued semantics so that whenever a $\mu$-calculus formula $\varphi$ has a definite value (true or false) on $M_i\!\uparrow$, the same value holds also for $M$. Thus, $\varphi$ can be checked on either $M_1\!\uparrow$ or $M_2\!\uparrow$ (or both), and if any of them returns a definite result, then this result holds also for $M$. If both checks result in an indefinite value, the composition of the components needs to be considered. However, instead of constructing the composition of $M_1\!\uparrow$ and $M_2\!\uparrow$, our approach identifies and composes only the parts of the components in which their composition is necessary in order to conclude the truth value of $\varphi$. It ignores the parts which can be handled separately. The resulting model is often significantly smaller than the full system.

We explain how our compositional approach can be combined with abstraction, in order to further reduce the size of the checked components. The result is an incremental compositional abstraction-refinement framework, which resembles automatic Assume-Guarantee reasoning.

## 1 Introduction

Model checking [11] is a useful approach for verifying properties of systems. The main disadvantage of model checking is the state explosion problem, which refers to its high space requirements. Two of the most promising approaches to fighting the state explosion problem are abstraction and compositional verification. In this work we join their forces to obtain a novel fully automatic compositional technique that can determine the truth value of the full $\mu$-calculus with respect to a given system.

In compositional model checking one tries to verify parts of the system separately in order to avoid the construction of the entire system. To account for the dependencies between the components, the Assume-Guarantee (AG) paradigm [22,25] suggests how to verify one module based on an *assumption* about the behavior of its environment, where the environment consists of the other system modules. The environment is then verified, in order to *guarantee* that it actually satisfies the assumption. Many of the works on compositional model checking are based on the AG paradigm and on learning [12,5,10] (see the related work section for more details). In contrast, our approach is

[⋆] The first author would like to acknowledge the financial support of an IBM Ph.D. Fellowship.

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 69–86, 2007.

based on techniques taken from the 3-valued game-based model checking for abstract models [26,18,19].

We first present our method for concrete systems, composed of concrete (unabstracted) components. We then extend it to abstract systems, in which one or both of the components have been abstracted (separately). For simplicity we refer to systems that consist of two components $M_1 \| M_2$. However, our approach can be extended to the composition of $n$ components. In our setting $M_1$ and $M_2$ are Kripke structures that synchronize on the joint labeling of the states. This composition is suitable for modeling synchronous systems with shared variables. In particular, it is suitable for hardware designs that synchronize on their inputs and outputs, since our models can be viewed as Moore machines [20]. The underlying ideas are applicable to other models as well, such as Labeled Transition Systems (LTSs), where components synchronize on their joint transitions and interleave their local transitions.

Given a system $M = M_1 \| M_2$, we view each component $M_i$ as an abstraction $M_i\!\uparrow$ of the global system $M$, in which the values of the local (unshared) variables and the transitions of the other component are unknown. We consider the 3-valued semantics of the $\mu$-calculus, in which the value of a formula in a model is either tt (true), ff (false), or $\perp$ (unknown). $M_i\!\uparrow$ is defined so that whenever a $\mu$-calculus formula $\varphi$ has a definite value (tt or ff) on $M_i\!\uparrow$, the same value holds also for $M$. Thus, $\varphi$ can be checked on either $M_1\!\uparrow$ or $M_2\!\uparrow$ (or both), and if any of them returns a definite result, then this result holds also for $M$. Only if both checks result in $\perp$, the value of $\varphi$ in $M$ is unknown.

For the 3-valued abstraction, when the model checking returns $\perp$, the abstract model should be *refined* in order to eliminate the $\perp$ result. For our framework, a refinement could be achieved by composing $M_1\!\uparrow$ and $M_2\!\uparrow$. This, however, is not desired and not necessary. Instead, only the parts of the abstract models for which the model checking result is $\perp$ are identified and composed. The resulting refined model is often significantly smaller than the full system and is guaranteed to return the correct model checking result.

More specifically, our approach is based on the 3-valued game for model checking of $\mu$-calculus, suggested in [18,19]. The game is played on a *game graph*, whose nodes are labeled by $s \vdash \psi$, where $s$ is a state in the checked model and $\psi$ is a subformula of the checked formula, s.t. the value of $\psi$ in $s$ is relevant for determining the model checking result. The model checking algorithm “colors” each node in the game graph by $T$, $F$, or $?$ iff the value of $\psi$ in $s$ is tt, ff or $\perp$, respectively. Recall that we first apply the model checking algorithm to each component separately. If the algorithm colors a node $s \vdash \psi$ of $M_1\!\uparrow$ with $T$ ($F$), then it is guaranteed that every state in the composed system $M$, whose first component is $s$, satisfies (falsifies) $\psi$. A similar property holds for $M_2\!\uparrow$. Thus, when the model checking returns $\perp$ then only the subgraphs of nodes whose color is $?$ require further checking and are therefore composed. As such, the game-based approach provides a natural way of identifying and focusing on the places where the value of the checked formula remained inconclusive.

To further reduce the size of the checked components, we combine our compositional approach with abstraction. Abstraction not only reduces the state-space of the components, but also allows to handle infinite-state components by abstracting them into finite-state components. Given a system composed of two (or more) components, we first abstract each component separately. However, in order to guarantee preservation of both tt and ff we require that the common alphabet (e.g. common inputs and outputs for hardware designs) will not be abstracted. Only local (unshared) variables can be abstracted. While this limits the amount of reduction that can be achieved by the abstraction on a single component, it enables additional reduction due to the compositional reasoning.

We propose an automatic construction of the initial abstraction for each component separately. We then proceed as before: we run a 3-valued model checking on each of the components. If both return $\perp$, we identify and compose the parts where indefinite results were obtained, and apply 3-valued model checking to the composed model. While in the concrete case this step always terminates with a definite result, here we may obtain an indefinite result due to abstraction. In such a case, we follow [26,18,19] in finding the *cause* for the indefinite result on the composed model. However, the refinement is applied on each of the components separately. Moreover, we adopt the incremental approach of [26,18,19] and refine only the indefinite part of each component.

An abstraction of a component $M_i$ (which comprises the environment of the other component) can be viewed as providing an assumption on $M_i$. From this point of view, when applying abstraction-refinement on one or both of the components, the result is an automatic mechanism for assumption generation, which is either symmetric (refers to both components) or asymmetric (abstracts only one component). In each iteration, more information about the component is revealed, by need – based on the cause for the indefinite result. This resembles iterative AG reasoning. The use of conservative abstractions guarantees that the assumption describes the component correctly (by construction). Thus unlike typical AG reasoning, this need not be verified.

In summary, our contribution is threefold:

- We introduce a new ingredient to compositional model checking, which enhances its modularity. Namely, given a compositional system, our approach uses a model checking game-graph as a means to identify and focus on the parts of the components in which their composition is indeed necessary to conclude the truth value of the checked property, due to dependencies between them. It uses the game-graph to exchange information between the components in these points, by need, and ignores the parts which can be handled separately. Thus, it avoids the construction of the full composition. Furthermore, if a certain formula only depends on one component, then it is resolved on this component alone while avoiding the composition altogether. Our technique is orthogonal to the AG approach, and can also be applied when the composed system consists of a component and an assumption on its environment.
- We develop a compositional, fully automatic, abstraction-refinement framework, which has some resemblance to iterative AG-reasoning, but benefits from the modular model checking described above. The refinement is also applied to each component separately. In addition, the abstraction-refinement is incremental in the sense that results from previous iterations are re-used. From the AG point of view, our compositional abstraction-refinement can be viewed as a new, automatic, mechanism for assumption generation, which uses the power of abstraction-refinement.

- Finally, unlike most automatic AG approaches, which are limited to universal safety properties, our technique is applicable to the *full* $\mu$-calculus.

**Related Work.** Recently, [12] followed by [5,10], considered automatic assumption generation for AG reasoning. They use *learning* algorithms for finite automata in order to automatically produce suitable assumptions for an AG rule. A similar approach is taken in [7], where the AG rule used is symmetric. Assumption generation in a more general setting is addressed in [16,2]. These works are all restricted to *universal safety* properties. The learning algorithms used in these works also perform some kind of an abstraction-refinement. However, these algorithms are not specifically tailored for verification. In particular, they do not always maintain a conservative abstraction of the environment. As such, the assumption sometimes needs to be weakened and sometimes needs to be strengthened. In our case an assumption (abstraction) should never be weakened. Moreover, we increase the modularity of the model checking step by using the game-based approach, which also enables an incremental analysis. Most importantly, our approach is applicable to the *full* $\mu$-calculus.

The game-based model checking enables us to identify the places where the value of a subformula in a component's state is the same for *all* environments. We exploit this information to reduce the model checking instance of the entire system. Other authors have also used similar information for reductions. In [1] the authors merge component's states that share the same value for a given CTL formula in all environments, thus minimizing the component. In [3] the authors use reachability and controllability information about the concrete components (gathered via game-theoretic techniques) in order to construct abstract components for *invariance* properties. The composition of the abstract components is then computed and model checked. We, on the other hand, do not try to minimize each component. Instead, the game-graph enables us to prune parts of each component's model checking instance whose effect was already taken into consideration. As a result, we reduce the state space exploration of the entire system. This is applicable even if no states of the individual components can be merged.

[15] uses controllability information to speed up falsification of invariance properties. They identify unpreventable violations of the property based on each component separately, which enables to prune the state space exploration of the compound system before a violation is actually encountered. The authors state that their method can be extended to arbitrary LTL properties. However, they only use controllability information w.r.t. the entire formula. Our approach enables to gather information about subformulas as well, and thus can result in more substantial reductions. In addition, our approach is aimed at both verification and falsification (with a 3-valued semantics) and is applicable to a full branching time logic.

[24] also uses 3-valued model checking for modular verification. They consider feature-oriented modules, where the composition is via interfaces and has a more sequential nature. As a result, they only refer to unknown propositions and not to uncertainty in the transitions. A substantial part of their work is devoted to determining what information needs to be included in a feature's interface to support compositional reasoning. In our case, we use the game graph for sharing such auxiliary information.

In [4] the authors suggest to use game structures to reason about composition of components. [14,6] suggest abstraction-refinement frameworks for such models, w.r.t.

alternating time temporal logics, which enable to describe properties of the interaction between components. We are interested in properties of the compound system, thus the focus in these works is different. In addition, they abstract each component separately and then model check the entire system. The model checking step is not modular.

[9] develops a compositional counterexample-guided abstraction refinement for a universal temporal logic (which extends ACTL). In their approach, the abstraction and the refinement steps are performed on each component separately, but the model checking step is done on the entire (abstract) system. In our approach, the model checking step is also compositional, and the properties considered are not limited to a universal logic.

## 2 Preliminaries

**$\mu$-calculus.** [23] Let $AP$ be a finite set of atomic propositions and $\mathcal{V}$ a set of propositional variables. The set of literals over $AP$ is $Lit = AP \cup \{\neg p : p \in AP\}$. We identify $\neg\neg p$ with $p$. The logic *$\mu$-calculus* in *negation normal form* over $AP$ is defined by:

$$\varphi \ ::= \ l \mid Z \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \Box\varphi \mid \Diamond\varphi \mid \mu Z.\varphi \mid \nu Z.\varphi$$

where $l \in Lit$ and $Z \in \mathcal{V}$. Intuitively, $\Box$ stands for "all successors", and $\Diamond$ stands for "exists a successor". $\mu$ denotes a least fixpoint, whereas $\nu$ denotes greatest fixpoint. We will also write $\eta$ for either $\mu$ or $\nu$. Let $\mathcal{L}_\mu$ denote the set of *closed* formulas generated by the above grammar, where the fixpoint quantifiers $\mu$ and $\nu$ are variable binders. We assume that formulas are well-named, i.e. no variable is bound more than once in any formula. Thus, every variable $Z$ *identifies* a unique subformula $fp(Z) = \eta Z.\psi$ of $\varphi$, where the set $Sub(\varphi)$ of *subformulas* of $\varphi$ is defined in the usual way.

**Concrete Semantics.** Concrete systems are typically modelled as *Kripke structures*. A Kripke structure [11] is a tuple $M = (AP, S, s^0, R, L)$, where $AP$ is a finite set of atomic propositions, $S$ is a finite set of states, $s^0 \in S$ is an initial state, $R \subseteq S \times S$ is a transition relation, and $L : S \rightarrow 2^{Lit}$ is a labeling function, such that for every state $s$ and every $p \in AP$, exactly one of $p$ and $\neg p$ is in $L(s)$.

The *concrete semantics* $[\![\varphi]\!]^M$ of a closed formula $\varphi \in \mathcal{L}_\mu$ over $AP$ w.r.t. a Kripke structure $M = (AP, S, s^0, R, L)$ is a mapping from $S$ to {tt, ff}. $[\![\varphi]\!]^M(s) =$ tt $(=$ ff) means that the formula $\varphi$ is true (false) in the state $s$ of the Kripke structure $M$. If $[\![\varphi]\!]^M(s^0) =$ tt $(=$ ff), we say that $M$ satisfies (falsifies) $\varphi$, denoted $M \models \varphi$ $(M \not\models \varphi)$.

**3-Valued Abstraction.** In the context of abstraction, *Kripke Modal Transition Systems* [21,17] are often used as abstract models that preserve the $\mu$-calculus.

**Definition 1.** *A* Kripke Modal Transition System *(***KMTS***) is a tuple $M = (AP, S, s^0, R^+, R^-, L)$, where $AP$, $S$ and $s^0$ are defined as before, $R^+, R^- \subseteq S \times S$ are must and may transition relations (resp.) such that $R^+ \subseteq R^-$, and $L : S \rightarrow 2^{Lit}$ is a labeling function such that for every state $s$ and $p \in AP$,* at most *one of $p$ and $\neg p$ is in $L(s)$.*

The *3-valued semantics* $[\![\varphi]\!]_3^M$ of a closed formula $\varphi \in \mathcal{L}_\mu$ w.r.t. a KMTS $M$ is a mapping from $S$ to $\{$tt, ff, $\perp\}$ [8,21]. It preserves both satisfaction (tt) and refutation (ff) from the abstract KMTS to the concrete model it represents. $\perp$ is a new truth value whose meaning is that the truth value over the concrete model is unknown and can be either tt or ff. The interesting cases in the definition of the 3-valued semantics are those of the literals and the modalities.

$$[\![l]\!]_3^M(s) = \text{ tt if } l \in L(s), \text{ ff if } \neg l \in L(s), \;\perp \text{ otherwise.}$$

$$[\![\Box\psi]\!]_3^M(s) = \begin{cases} \text{tt, if } \forall t \in S, \text{ if } sR^-t \text{ then } [\![\psi]\!]_3^M(t) = \text{tt} \\ \text{ff, if } \exists t \in S \text{ s.t. } sR^+t \text{ and } [\![\psi]\!]_3^M(t) = \text{ff} \\ \perp, \text{otherwise} \end{cases}$$

and dually for $\Diamond\psi$ when exchanging tt and ff. The notations $M \models \varphi$ and $M \not\models \varphi$ are used for KMTSs as well. In addition, if $[\![\varphi]\!]_3^M(s^0) = \perp$, the value of $\varphi$ in $M$ is indefinite.

The following definition formalizes the relation between two KMTSs that guarantees preservation of $\mu$-calculus formulas w.r.t. the 3-valued semantics.

**Definition 2 (Mixed Simulation).** *[13,17] Let* $M_1 = (AP, S_1, s_1^0, R_1^+, R_1^-, L_1)$ *and* $M_2 = (AP, S_2, s_2^0, R_2^+, R_2^-, L_2)$ *be two KMTSs, both defined over AP.* $H \subseteq S_1 \times S_2$ *is a* mixed simulation *from* $M_1$ *to* $M_2$ *if* $(s_1, s_2) \in H$ *implies:*

1. $L_2(s_2) \subseteq L_1(s_1)$.
2. *if* $s_1 R_1^- s_1'$*, then there is some* $s_2' \in S_2$ *such that* $s_2 R_2^- s_2'$ *and* $(s_1', s_2') \in H$.
3. *if* $s_2 R_2^+ s_2'$*, then there is some* $s_1' \in S_1$ *such that* $s_1 R_1^+ s_1'$ *and* $(s_1', s_2') \in H$.

*If there is a mixed simulation H s.t.* $(s_1^0, s_2^0) \in H$*, then* $M_2$ abstracts $M_1$*, denoted* $M_1 \preceq M_2$.

In particular, Def. 2 can be applied to a (concrete) Kripke structure $M_C$ and an (abstract) KMTS $M_A$, by viewing the Kripke structure as a KMTS where $R^+ = R^- = R$. For a Kripke structure, the 3-valued semantics agrees with the concrete semantics. Thus, preservation of $\mathcal{L}_\mu$ formulas is guaranteed by the following theorem.

**Theorem 1.** *[17] Let* $H \subseteq S_1 \times S_2$ *be the mixed simulation relation from a KMTS* $M_1$ *to a KMTS* $M_2$*. Then for every* $(s_1, s_2) \in H$ *and every* $\varphi \in \mathcal{L}_\mu$ *we have that* $[\![\varphi]\!]_3^{M_2}(s_2) \neq \perp \Rightarrow [\![\varphi]\!]_3^{M_1}(s_1) = [\![\varphi]\!]_3^{M_2}(s_2)$.

**Abstract Model Checking.** A 3-valued game-based model checking for the $\mu$-calculus over KMTSs was suggested in [18,19]. They introduce 3-valued parity games and translate the 3-valued model checking problem into the problem of determining the winner in a 3-valued satisfaction game, which is a special case of a 3-valued parity game. We omit the details of the 3-valued satisfaction game, but continue with the *game graph*, which presents all the information "relevant" for the model checking.

**Game Graph.** Let $M = (AP, S, s^0, R^+, R^-, L)$ be a KMTS and $\varphi \in \mathcal{L}_\mu^0$. The *game graph* $G_{M\times\varphi}$, or in short $G$, is a graph $(N, n^0, E^+, E^-)$ where $N \subseteq S \times Sub(\varphi)$ is a set of nodes and $E^+ \subseteq E^- \subseteq N \times N$ are sets of must and may edges defined as follows. $n^0 = s^0 \vdash \varphi \in N$ is the initial node. The (rest of the) nodes and the edges are defined by the rules of Fig. 1, with the meaning that whenever $n \in N$ is of the form of the upper part of the rule, the result in the lower part of the rule is also a node $n' \in N$ and $E^-(n, n')$. Moreover, $E^+(n, n')$ holds as well in all cases except for an application of the rules in the second column with a model's transition $(s, t) \in R^- \setminus R^+$. Intuitively, the outgoing edges of $s \vdash \psi \in N$ define "subgoals" for checking $\psi$ in $s$.

$$\begin{array}{lll}
\dfrac{s\vdash\psi_0\vee\psi_1}{s\vdash\psi_i} : i\in\{0,1\} & \dfrac{s\vdash\Diamond\psi}{t\vdash\psi} :\ sR^{+}t \text{ or } sR^{-}t & \dfrac{s\vdash\eta Z.\psi}{s\vdash Z} \\
\dfrac{s\vdash\psi_0\wedge\psi_1}{s\vdash\psi_i} : i\in\{0,1\} & \dfrac{s\vdash\Box\psi}{t\vdash\psi} :\ sR^{+}t \text{ or } sR^{-}t & \dfrac{s\vdash Z}{s\vdash\psi} : \text{if } fp(Z)=\eta Z.\psi
\end{array}$$

**Fig. 1.** Rules for game graph construction

If $E^{-}(n,n')$ $(E^{+}(n,n'))$ then $n'$ is a may (must) son of $n$. A node $n = s\vdash\psi$ in $G_{M\times\varphi}$ is classified as a $\wedge$, $\vee$, $\Box$, $\Diamond$ node, based on $\psi$. If $\psi$ is of the form $Z$ or $\eta Z.\psi'$, $n$ is *deterministic* – it has exactly one son. If $n$ has no outgoing edges, then it is a *terminal node*. In a full game graph this means that either $\psi$ is a literal, or $\psi$ is of the form $\Diamond\psi'$ or $\Box\psi'$, and $s$ has no outgoing transition in $M$.

Fig. 2(b) presents examples of game graphs for $\varphi = \Box(\neg i\vee\Diamond o)$ and the models from Fig. 2(a), where all transitions are considered may transitions.

**Coloring Algorithm.** The model checking algorithms of [18,19] can be viewed as coloring algorithms that label (color) each node $n = s\vdash\psi$ in the game graph by $T$, $F$, ? depending on the truth value of $\psi$ in the state $s$ in $M$ (based on the 3-valued semantics). The result of the coloring is a *3-valued coloring function* $\chi : N\to\{T,F,?\}$.

In both cases the coloring is performed by solving the 3-valued parity game for satisfaction, where each color stands for a possible result in the game. The algorithm of [18] is a generalization of Zielonka's algorithm for solving (2-valued) parity games. In [19], the 3-valued satisfaction game is reduced into two (2-valued) parity games, improving the coloring's complexity. The following formalizes the correctness of the coloring. For a (possibly not closed) formula $\psi$, $\psi^{*}$ denotes the result of replacing every free occurrence of $Z\in\mathcal{V}$ in $\psi$ by $fp(Z)$. Note that if $\psi$ is closed, then $\psi^{*}=\psi$.

**Definition 3.** *Let $G_{M\times\varphi}$ be a game graph for a KMTS $M$ and $\varphi\in\mathcal{L}_{\mu}$. A (possibly partial) coloring function $\chi : N\to\{T,F,?\}$ for $G_{M\times\varphi}$ (or its subgraph) is* correct *if for every $s\vdash\psi\in N$, whenever $\chi(s\vdash\psi)$ is defined, then:*

1. $[\![\psi^{*}]\!]_{3}^{M}(s) = tt$ *iff* $\chi(s\vdash\psi) = T$.
2. $[\![\psi^{*}]\!]_{3}^{M}(s) = ff$ *iff* $\chi(s\vdash\psi) = F$.
3. $[\![\psi^{*}]\!]_{3}^{M}(s) = \perp$ *iff* $\chi(s\vdash\psi) = ?$.

**Theorem 2.** *[18,19] Let $\chi_F$ be the (total) coloring function returned by the coloring algorithm of [18] or [19] for $G_{M\times\varphi}$. Then $\chi_F$ is correct.*

Moreover, in both cases, the final coloring of the nodes reflects the 3-valued semantics of the logic: A $\wedge$-node or a $\Box$-node is colored $T$ iff all its may sons are colored $T$ (and in particular if it has no may sons), it is colored $F$ iff it has a must son which is colored $F$, and otherwise it is colored ?. Dually for a $\vee$-node or a $\Diamond$-node when exchanging $T$ and $F$. The color of $s\vdash l$ for $l\in Lit$ is $T$ iff $l\in L(s)$, $F$ iff $\neg l\in L(s)$, and ? otherwise. The result of the coloring is demonstrated in Fig. 2(b).

**Refinement.** If the model checking result of an abstract model is indefinite ($\perp$), a refinement is needed. When using the coloring algorithms of [18,19], an indefinite result is accompanied with a *failure state* and a *failure cause*. The failure cause is either a literal whose value in the failure state is $\perp$, or an outgoing may transition of the failure state in the underlying model which is not a must transition. Refinement is then performed by splitting the abstract states in a way that eliminates the failure cause (see [18,19]).

## 3 Partial Coloring and Subgraphs

In the following sections we use the game-based model checking in order to identify and focus on the places where the dependencies between components of the system affect the model checking result. In this section we set the basis for this, by investigating properties of the game graph and the coloring algorithms.

Due to their nature, as algorithms for solving a 3-valued parity game, the coloring algorithms of [18,19] have the important property that they can be applied on a partially colored graph, in which case they extend the given coloring to the rest of the graph in a correct way. Moreover, the coloring can also be applied on a partially colored *subgraph*, and under certain assumptions it will yield a correct coloring of the subgraph. To formalize this, we need the following definitions.

**Definition 4.** *Let $G$ be a game graph and $\chi_F$ its final coloring function. For a non-terminal node $n$ in $G$ we define its* witnessing sons *as follows, depending on its type:*

**$\wedge, \Box$:** *the witnessing sons are those colored $F$ or $?$ by $\chi_F$.*
**$\vee, \Diamond$:** *the witnessing sons are those colored $T$ or $?$ by $\chi_F$.*
**deterministic:** *the witnessing son is the only son.*

The sons are witnessing in the sense that they suffice to determine the color of the node, thus removing the rest of the node's sons from the graph does not damage the result of the coloring. For example, if a $\wedge$-node or a $\Box$-node has no witnessing sons, meaning all its sons are colored $T$, then we know it should be colored $T$, and this is indeed how the coloring algorithms will color the node when keeping only the witnessing sons. Otherwise, the witnessing sons determine whether the node should be colored $F$ or $?$, thus one can correctly color the node by considering only them.

**Definition 5.** *A subgraph $G'$ of a game graph $G$ is* closed *if every node in $G'$ is either a terminal node, or* all *its witnessing sons (and corresp. edges) from $G$ are also in $G'$.*

**Theorem 3.** *Consider a closed subgraph $G'$ of a game graph $G$ with a partial coloring function $\chi$ which is correct and defined over (at least) all the terminal nodes in $G'$. Then applying the coloring algorithm of [18] or [19] on $G'$ with $\chi$ as an initial coloring results in a correct coloring of $G'$.*

In fact, for the coloring of the subgraph to be correct, not *all* the witnessing sons are needed, as long as there is enough information to explain the correct coloring of each uncolored node. However, we will see that in our case we will need all of them, as we will deduce from the game graph of one component to the game graph of the full system, where some of the nodes will be removed and for some an indefinite color (?)

will change into $T$ or $F$. This means that some of the witnessing sons will not remain witnessing sons in the game graph of the full system. Thus, we will not be able to know a-priori which of them is the "right" choice to include in a way that will also provide the necessary information for a correct coloring in the game graph of the full system.

Another notion that we will need later is the following.

**Definition 6 (?-Subgraph).** *Let $G$ be a colored graph whose initial node is colored $?$. The* $?$-subgraph *is the least subgraph $G_?$ of $G$ that obeys the following:*

- *the initial node is in $G_?$ (and is the initial node of $G_?$).*
- *For each node in $G_?$ which is colored $?$ in $G$ all its witnessing sons (and corresponding edges) in $G$ are included in $G_?$.*

*$G_?$ is accompanied with a partial coloring function $\chi_I$ which is defined over the terminal nodes in $G_?$, and colors them as the coloring function $\chi_F$ of $G$.*

The $?$-subgraph $G_?$ and its initial coloring meet the conditions of Thm. 3. Intuitively, this means that $G_?$ contains *all* the information regarding the indefinite result. Fig. 2(b) provides examples of $?$-subgraphs.

## 4 Compositional Model Checking

In compositional model checking the goal is to verify a formula $\varphi$ on a compound system $M_1 \| M_2$. In our setting $M_1$ and $M_2$ are Kripke structures that synchronize on the joint labelling of the states. Since a Kripke structure is a special case of a KMTS where $R = R^+ = R^-$, we define the composition for the more general case of KMTSs. In the following we denote by $Lit_1$ and $Lit_2$ the sets of literals over $AP_1$ and $AP_2$, resp.

**Definition 7.** *Two KMTSs $M_1 = (AP_1, S_1, s_1^0, R_1^+, R_1^-, L_1)$ and $M_2 = (AP_2, S_2, s_2^0, R_2^+, R_2^-, L_2)$ are* composable *if their initial states agree on their joint labeling, i.e. $L_1(s_1^0) \cap Lit_2 = L_2(s_2^0) \cap Lit_1$.*

**Definition 8.** *Let $M_1 = (AP_1, S_1, s_1^0, R_1^+, R_1^-, L_1)$ and $M_2 = (AP_2, S_2, s_2^0, R_2^+, R_2^-, L_2)$ be two composable KMTSs. We define their composition, denoted $M_1 \| M_2$, to be the KMTS $(AP, S, s^0, R^+, R^-, L)$, where*

- $AP = AP_1 \cup AP_2$
- $S = \{(s_1, s_2) \in S_1 \times S_2 \mid L_1(s_1) \cap Lit_2 = L_2(s_2) \cap Lit_1\}$
- $s^0 = (s_1^0, s_2^0)$
- $R^+ = \{((s_1, s_2), (t_1, t_2)) \in S \times S \mid (s_1, t_1) \in R_1^+ \text{ and } (s_2, t_2) \in R_2^+\}$
- $R^- = \{((s_1, s_2), (t_1, t_2)) \in S \times S \mid (s_1, t_1) \in R_1^- \text{ and } (s_2, t_2) \in R_2^-\}$
- $L((s_1, s_2)) = L(s_1) \cup L(s_2)$

*In particular, if $M_1$ and $M_2$ are Kripke structures with transition relations $R_1$ and $R_2$ resp., then $M_1 \| M_2$ is a Kripke structure with $R = \{((s_1, s_2), (t_1, t_2)) \in S \times S \mid (s_1, t_1) \in R_1 \text{ and } (s_2, t_2) \in R_2\}$.*

From now on we fix $AP$ to be $AP_1 \cup AP_2$. For $i \in \{1,2\}$ we use $\overline{i}$ to denote the remaining index in $\{1,2\} \setminus \{i\}$.

We use the mechanism produced for abstractions of full branching time logics for the purpose of compositional verification. The basic idea is to view each Kripke structure $M_i$ as a partial model that abstracts $M_1 \| M_2$.

**Definition 9.** *Let $M_i = (AP_i, S_i, s_i^0, R_i, L_i)$ be a Kripke structure. We lift $M_i$ into a KMTS $M_i\!\uparrow = (AP, S_i, s_i^0, R_i^+\!\uparrow, R_i^-\!\uparrow, L_i\!\uparrow)$ over $AP$ where $R_i^+\!\uparrow = \emptyset$, $R_i^-\!\uparrow = R_i$ and $L_i\!\uparrow(s) = L_i(s)$.*

That is, we view $M_i$ as a KMTS $M_i\!\uparrow$ over $AP$ (rather than $AP_i$). This immediately makes the value of each literal over $AP \setminus AP_i$ in each state of $M_i\!\uparrow$ indefinite (as neither $p$ nor $\neg p$ are in $L_i(s)$) – indeed, it depends on $M_{\overline{i}}$. In addition, each transition of $M_i$ is considered a may transition (since in the composition it might be removed if a matching transition does not exist in $M_{\overline{i}}$, but transitions can never be added).

**Theorem 4.** $M_1 \| M_2 \preceq M_i\!\uparrow$. *The mixed simulation is* $\{((s_1, s_2), s_i) \mid (s_1, s_2) \in S\}$.

Since each $M_i\!\uparrow$ abstracts $M_1 \| M_2$, we are able to first consider each component separately: Thm. 1 ensures that if $\varphi$ has a definite value (tt or ff) in $M_i\!\uparrow$ under the 3-valued semantics, then the same value holds in $M_1 \| M_2$ as well. In particular, the values in $M_1\!\uparrow$ and $M_2\!\uparrow$ cannot be contradictory, and it suffices that one of them is definite in order to determine the value in $M_1 \| M_2$.

The more typical case is that the value of $\varphi$ on both $M_1\!\uparrow$ and $M_2\!\uparrow$ is indefinite. This reflects the fact that $\varphi$ depends on both components and their synchronization. Typically, an indefinite result requires some refinement of the abstract model. In our case refinement means considering the composition with the other component. Still, in this case as well, having considered each component separately can guide us into focusing on the places where we indeed need to consider the composition of the components.

The game-based approach to model checking provides a convenient way for presenting this information. If the KMTS $M_i\!\uparrow$ is model checked using the algorithm of [18] or [19], then the result is a colored game graph, in which $T$ and $F$ represent definite results (i.e. truth values that hold no matter what the environment is), but the $?$ color needs to be resolved by considering the composition. This is where the $?$-subgraph (see Def. 6) becomes handy, as it points out the places where this is really needed.

The $?$-subgraph for each component is computed top-down, starting from the initial node. As long as a node colored $?$ is encountered, the search continues in a BFS manner by including the witnessing sons. Definite nodes which are included in the subgraph become terminal nodes, and their coloring defines the initial coloring function.

The $?$-subgraphs of the two colored graphs present all the indefinite information that results from the dependencies between the components. Thus, to resolve the indefinite result, we compose the $?$-subgraphs.

**Definition 10 (Product Graph).** *Let $G_{?_1}$ and $G_{?_2}$ be two $?$-subgraphs as above with initial nodes $s_1^0 \vdash \varphi$ and $s_2^0 \vdash \varphi$ resp. We define their product to be the least graph $G_\| = (N_\|, n_\|^0, E_\|^+, E_\|^-)$ such that:*

- $n_{\|}^0 = (s_1^0, s_2^0) \vdash \varphi$ *is the initial node in* $N_{\|}$.
- *If* $(s_1, s_2) \vdash \psi \in N_{\|}$ *and* $(s_1 \vdash \psi, s_1' \vdash \psi') \in E_1^-$ *and* $(s_2 \vdash \psi, s_2' \vdash \psi') \in E_2^-$ *and* $L_1(s_1') \cap Lit_2 = L_2(s_2') \cap Lit_1$ *(i.e.* $(s_1', s_2')$ *is a state of* $M_1 \| M_2$*), then:* $(s_1', s_2') \vdash \psi' \in N_{\|}$ *and* $((s_1, s_2) \vdash \psi, (s_1', s_2') \vdash \psi')$ *is in* $E_{\|}^+$ *and* $E_{\|}^-$.

Note that all the edges in $G_{\|}$ are must edges, whereas in the ?-subgraphs we had may edges (the transitions of each component were treated as may transitions in the lifted version). This is because the product graph already refers to the complete system $M_1 \| M_2$, where all transitions are concrete transitions (modeled as must transitions).

The product graph is constructed by a top-down traversal of the subgraphs, where, starting from the initial nodes, nodes that share the same formulas and whose states agree on the joint labeling are composed (recall that $s_1^0$ and $s_2^0$ agree on their joint labeling). Whenever two non-terminal nodes are composed, the outgoing edges are computed as the product of their outgoing edges, limited to legal nodes (w.r.t. the restriction to states that agree on their labeling). In particular, this means that if a node in one subgraph has no matching node in the other, then it will be omitted from the product graph. In addition, when a terminal node of one subgraph is composed with a non-terminal node of the other, the resulting node is a terminal node in $G_{\|}$.

We accompany $G_{\|}$ with an initial coloring function for its terminal nodes based on the initial coloring functions of the two subgraphs. We use the following observation:

**Proposition 1.** *Let* $n = (s_1, s_2) \vdash \psi$ *be a terminal node in* $G_{\|}$*. Then one of the following holds. Either (a) at least one of* $s_1 \vdash \psi$ *and* $s_2 \vdash \psi$ *is a terminal node in its subgraph, in which case at least one of them is colored by a definite color by the initial coloring of its subgraph, and contradictory definite colors are impossible. We denote this color by* $col(n)$*; Or (b) both* $s_1 \vdash \psi$ *and* $s_2 \vdash \psi$ *are non-terminal nodes but no outgoing edges were left in their composition.*

**Definition 11.** *We define the initial coloring function* $\chi_I$ *of* $G_{\|}$ *as follows. Let* $n$ *be a terminal node in* $N_{\|}$*. If it fulfills case (a) of Prop. 1, then* $\chi_I(n) = col(n)$*. If it fulfills case (b), then* $\chi_I(n) = T$ *if* $n$ *is a* $\wedge$*-node or a* $\Box$*-node, and* $\chi_I(n) = F$ *if* $n$ *is a* $\vee$*-node or a* $\Diamond$*-node.* $\chi_I$ *is undefined for the rest of the nodes.*

In particular, if a terminal node in $G_{\|}$ results from a terminal node which is colored by ? in one subgraph and a terminal node which is colored by some definite color in the other, then the definite color takes over.

Note that the initial coloring function of the product graph colors all the terminal nodes by definite colors. Along with the property that all the edges in the product graph are must edges, this reflects the fact that the composition resolves all the indefinite information that existed in each component when it was considered separately. Therefore, when applying (one of) the coloring algorithm to the product graph, all the nodes are colored by definite colors (in fact, a 2-valued coloring can be applied).

**Theorem 5.** *The resulting product graph* $G_{\|}$ *is a closed subgraph of the game graph over* $M_1 \| M_2$*. In addition, the initial coloring function is correct w.r.t.* $M_1 \| M_2$ *and defined over all the terminal nodes in the subgraph.*

By Thm. 3, this means that coloring $G_{\|}$ results in a correct result w.r.t. the model checking of $\varphi$ in $M_1 \| M_2$. Thus, to model check $\varphi$ on $M_1 \| M_2$ it remains to color $G_{\|}$. Note that the full graph for $M_1 \| M_2$ is not constructed. To sum up, the algorithm is as follows.

**Step 1** Model check each $M_i\!\uparrow$ separately (for $i \in \{1, 2\}$):

1. Construct the game graph $G_i$ for $\varphi$ and $M_i\!\uparrow$.
2. Apply the 3-valued coloring on $G_i$. Let $\chi_i$ be the resulting coloring function.

If $\chi_1(n_1^0)$ or $\chi_2(n_2^0)$ is definite, return the corresp.model checking result for $M_1 \| M_2$.

**Step 2** Consider the composition $M_1 \| M_2$:

1. Construct the ?-subgraphs for $G_1$ and $G_2$.
2. Construct the product graph $G_{\|}$ of the ?-subgraphs.
3. Apply the 3-valued coloring on $G_{\|}$ (with the initial coloring function).

Return the model checking result corresponding to $\chi_{\|}(n_{\|}^0)$.

*Example 1.* Consider the components depicted in Fig. 2(a). The atomic proposition $o$ (short for $output$) is local to $M_1$, $i$ ($input$) is local to $M_2$, and $r$ ($receive$) is the only joint atomic proposition that $M_1$ and $M_2$ synchronize on. Suppose we wish to verify in $M_1 \| M_2$ the property $\Box(\neg i \vee \Diamond o)$, which states that in all the successor states of the initial state, an $input$ signal implies that there is a successor state where the $output$ signal holds. Fig. 2(b) depicts the colored game graph of each (lifted) component, and highlights the ?-subgraph of each of them. The product graph and its coloring is depicted in Fig. 2(c), as an "intersection" of the two subgraphs. All the edges in the product graph are must edges. All nodes, and in particular the initial node, are colored $T$, thus the property is verified. One can see that most of the efforts were done on each component separately, and the product graph only considers a small part of the compound system.

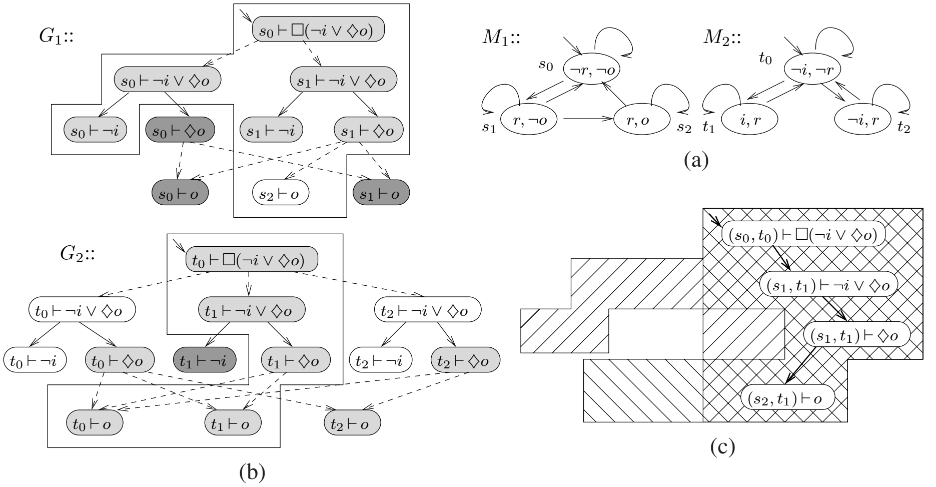


**Fig. 2.** (a) Components, (b) their game graphs and their ?-subgraphs (enclosed by a line), and (c) the product graph. Dashed edges denote may edges which are not must edges. The colors reflect the coloring function: white stands for $T$, dark gray stands for $F$ and light gray stands for ?.

## 5 Adding Abstraction

In Section 4 we considered concrete components. The indefinite results on each component resulted only from their interaction, and were resolved by composing the indefinite parts. We now combine this idea with existing abstraction-refinement techniques.

### 5.1 Motivation

Composing the ?-subgraphs of two components, as suggested in Section 4, corresponds to refining *all* possible failure causes. We now show how to use abstraction in order to make the refinement more local and gradual by eliminating *one* failure cause at a time.

Suppose that the coloring of the game-graph $G_1$ for the lifted concrete component $M_1\!\uparrow$ results in an indefinite result. We wish to eliminate the failure cause returned by the coloring algorithm for $M_1\!\uparrow$. Suppose that $s$ is the failure state. It abstracts all the states of $M_1 \| M_2$ that consist of $s$ and a matching state of $M_2$. Eliminating the cause for failure amounts to exposing from $M_2$ the information that involves the failure, and splitting $s$ accordingly. For example, in Fig. 2, a possible failure cause in $G_1$ is the may transition of $M_1\!\uparrow$ from $s_1$ to $s_2$. In order to either remove it or turn it into a must transition, we need to consider all the states of $M_2$ which are composable with $s_1$. These are the states labeled $r$. We need to find out which of them have a transition to a state labeled $r$ (i.e., a state composable with $s_2$), and which of them do not.

Clearly, the complete composition of the ?-subgraphs achieves this goal. However, it exposes more information than relevant for the given failure cause. Thus we do not want to resort to that (in this example it is indeed necessary, but in the general case not all the causes for failure need to be eliminated). We now sketch the idea that allows us to only consider the information from $M_2$ that is needed for eliminating the failure cause of $M_1\!\uparrow$. This will be described more formally in Section 5.2.

We abstract $M_2$ into $\hat{M}_2$. We start with a most coarse abstraction of $M_2$ w.r.t. $AP_1 \cap AP_2$, where each state is abstracted by its labeling, restricted to $AP_1 \cap AP_2$.

**Definition 12.** *Let $M_i = (AP_i, S_i, s_i^0, R_i, L_i)$ be a Kripke structure. The* most coarse abstraction *for $M_i$ w.r.t. $AP' \subseteq AP_i$ is the KMTS $\hat{M}_i^* = (AP_i, 2^{AP'}, L_i(s_i^0) \cap AP', \emptyset, 2^{AP'} \times 2^{AP'}, L_i^*)$, where for $\hat{s} \in 2^{AP'}$, $L_i^*(\hat{s}) = \hat{s} \cup \{\neg p \mid p \in AP' \setminus \hat{s}\}$.*

**Theorem 6.** *$M_i \preceq \hat{M}_i^*$. The mixed simulation is $\{(s_i, L_i(s_i) \cap AP') \mid s_i \in S_i\}$.*

The construction of the most coarse abstraction requires almost no knowledge of the component. More precise transitions can be computed as in [26]. Starting from the most coarse abstraction of $M_2$, we iteratively model check the composition of $M_1$ and the abstract model $\hat{M}_2$. The model checking is performed in a compositional fashion, similarly to Section 4, without computing the full composition. If the result in some iteration is indefinite, we refine $\hat{M}_2$ depending on the failure cause over $M_1 \| \hat{M}_2$. Recall that our purpose was to eliminate a failure cause over $M_1\!\uparrow$. Since we start with a most coarse abstraction of $M_2$ w.r.t. the joint atomic propositions, $M_1 \| \hat{M}_2$ is initially isomorphic to $M_1\!\uparrow$. As a result, in the first iteration the failure cause over $M_1 \| \hat{M}_2$ reflects the failure cause over $M_1\!\uparrow$, and the refinement of $\hat{M}_2$ indeed exposes the relevant information from $M_2$. Similarly, in the next iterations, the failure cause over $M_1 \| \hat{M}_2$ reflects the failure cause over $M_1\!\uparrow$, after taking into consideration the elimination of previous failure causes. In this sense, in each iteration we eliminate one failure cause over $M_1\!\uparrow$, and $\hat{M}_2$ "accumulates" the information required to eliminate these failure causes.

This means that we keep one of the components, $M_1$, concrete, and construct an abstract environment for it, by applying an iterative abstraction-refinement on $M_2$, where refinement is aimed at eliminating the indefinite results that arise when considering $M_1$ with the abstract environment. This approach is reminiscent of an asymmetric Assume-Guarantee rule. The next step is to make the approach symmetric by abstracting both components. This amounts to constructing abstract environments for both the components. In this case, refinement also needs to be applied on both components.

### 5.2 Compositional Abstraction-Refinement

We now describe in detail the combination of the compositional approach with abstraction-refinement. This provides a framework for using both the asymmetric and the symmetric approaches sketched above. On the one hand, we enhance the compositional model checking of Section 4 by using abstraction and a more gradual refinement. On the other hand, we enhance the abstraction-refinement framework by making both the abstract model checking and the refinement compositional. We no longer require that the state spaces of the concrete components are finite, as long as the abstract state spaces are.

**Compositional Abstraction.** Composition of abstract models (KMTSs) is defined in Def. 8. In order to ensure that the composition of two abstract models $\hat{M}_1 = (AP_1, \hat{S}_1, \hat{s}^0_1, R^+_1, R^-_1, \hat{L}_1)$ and $\hat{M}_2 = (AP_2, \hat{S}_2, \hat{s}^0_2, R^+_2, R^-_2, \hat{L}_2)$, for $M_1$ and $M_2$ respectively, results in an abstract model for $M_1 \| M_2$, we consider *appropriate* abstract models w.r.t. $AP_1 \cap AP_2$. We say that $\hat{M}_i$ is an *appropriate* abstract model of $M_i$ w.r.t. $AP_1 \cap AP_2$ if $\hat{M}_i$ and $M_i$ are related by a mixed simulation relation which is appropriate w.r.t. $AP_1 \cap AP_2$, as defined below.

**Definition 13.** *Let $H \subseteq S_i \times \hat{S}_i$ be a mixed simulation from $M_i$ to $\hat{M}_i$, both defined over $AP_i$. We say that $H$ is* appropriate *w.r.t. $AP' \subseteq AP_i$ if for every $(s_i, \hat{s}_i) \in H$, $L_i(s_i) \cap Lit' = \hat{L}_i(\hat{s}_i) \cap Lit'$, where $Lit'$ denotes the set of literals over $AP'$.*

In particular, the most coarse abstraction w.r.t. $AP_1 \cap AP_2$ (see Def. 12) is appropriate w.r.t. $AP_1 \cap AP_2$. Appropriateness of $\hat{M}_1$ and $\hat{M}_2$ w.r.t. $AP_1 \cap AP_2$ means that the abstraction of each component only identifies states that agree on their labelings w.r.t. the joint atomic propositions. It ensures that if $(\hat{s}_1, \hat{s}_2)$ is a state of the abstract composition and $\hat{s}_1$ abstracts $s_1$ and $\hat{s}_2$ abstracts $s_2$, then since $\hat{s}_1$ and $\hat{s}_2$ agree on the joint labeling, then so do $s_1$ and $s_2$. This ensures that $(s_1, s_2)$ is a state of the concrete composition, abstracted by $(\hat{s}_1, \hat{s}_2)$. We now have the following.

**Theorem 7.** *Let $\hat{M}_i$ be an appropriate abstract model for $M_i$ w.r.t. $AP_1 \cap AP_2$. Then $M_1 \| M_2 \preceq \hat{M}_1 \| \hat{M}_2$.*

Thus, if each of $M_1$ and $M_2$ is abstracted separately by an appropriate abstraction w.r.t. $AP_1 \cap AP_2$, then the composition of the corresponding abstract components $\hat{M}_1$ and $\hat{M}_2$ results in an abstract model for $M_1 \| M_2$. However, we do not wish to construct $\hat{M}_1 \| \hat{M}_2$ and model check it. Instead, we suggest to model check $\hat{M}_1 \| \hat{M}_2$ compositionally.

**Compositional (abstract) Model Checking.** The general scheme is similar to the concrete case: we first try to make the most out of each (abstract) component separately, and if this does not result in a definite answer, we consider the product of the ?-subgraphs which enable to exchange information via a compact representation. We start by viewing each abstract component $\hat{M}_i$ as a partial model that abstracts their composition $\hat{M}_1 \| \hat{M}_2$.

**Definition 14.** *Let $\hat{M}_i = (AP_i, \hat{S}_i, \hat{s}^0_i, R^+_i, R^-_i, \hat{L}_i)$ be a KMTS. We lift $\hat{M}_i$ into a KMTS $\hat{M}_i \!\uparrow = (AP, \hat{S}_i, \hat{s}^0_i, R^+_i \!\uparrow, R^-_i \!\uparrow, \hat{L}_i \!\uparrow)$ over $AP$ where $R^+_i \!\uparrow = \emptyset$, $R^-_i \!\uparrow = R^-_i$ and $\hat{L}_i\!\uparrow(\hat{s}) = \hat{L}_i(\hat{s})$.*

That is, when $\hat{M}_i$ is lifted into $\hat{M}_i\!\uparrow$, only the may transitions of $\hat{M}_i$ are useful, because must transitions are not really must w.r.t. $\hat{M}_1 \| \hat{M}_2$. Similarly to the concrete case:

**Theorem 8.** $\hat{M}_1 \| \hat{M}_2 \preceq \hat{M}_i\!\uparrow$.

**Corollary 1.** *If $\hat{M}_i$ is an appropriate abstract model for $M_i$ w.r.t. $AP_1 \cap AP_2$, then $M_1 \| M_2 \preceq \hat{M}_i\!\uparrow$.*

Therefore one can model check each of $\hat{M}_i\!\uparrow$ separately, and the definite results follow through to $M_1 \| M_2$. In fact, it is possible to show that $M_1 \| M_2 \preceq \hat{M}_i\!\uparrow$ holds even if we omit the appropriateness requirement. Thus appropriateness is not needed for this step. However, it is needed for the next steps, where we deduce from $\hat{M}_1 \| \hat{M}_2$ to $M_1 \| M_2$.

If both checks result in indefinite results, the (abstract) ?-subgraphs for both game graphs are produced and their product is considered. Having composed the ?-subgraphs of the two components resolves dependencies between them, but the result is still abstract, as it refers to the *abstract* composition $\hat{M}_1 \| \hat{M}_2$. This results in two differences compared to the concrete case.

First, the may edges do not necessarily become must edges. Instead, the distinction between may and must edges is determined by the type of the underlying transitions in the (unlifted) abstract models $\hat{M}_i$, which have been ignored so far. Second, it is now possible that a terminal node $n = (\hat{s}_1, \hat{s}_2) \vdash \psi$ in $G_\|$ with $\psi = l$ for a local literal $l \in Lit \setminus (Lit_1 \cap Lit_2)$ results from terminal nodes $\hat{s}_1 \vdash l$ and $\hat{s}_2 \vdash l$ which are *both* colored by ? in their subgraphs (one, since $l$ is local to the other component, and is thus treated as indefinite, and the other due to the abstraction). We add this possibility as case (c) to Prop. 1 which characterizes the terminal nodes in the product graph $G_\|$. It is taken into account when determining the initial coloring of $G_\|$.

**Definition 15 (Abstract Product Graph).** *Let $G_{?1}$ and $G_{?2}$ be two abstract ?-subgraphs as above. Their product graph $G_\| = (N_\|, n^0_\|, E^+_\|, E^-_\|)$ is defined as before, except for the definition of $E^+_\|$: an edge $((\hat{s}_1, \hat{s}_2) \vdash \psi, (\hat{s}'_1, \hat{s}'_2) \vdash \psi')$ in $E^-_\|$ is also in $E^+_\|$ iff $\hat{s}_i R^+_i \hat{s}'_i$ for each $i \in \{1, 2\}$. The initial coloring function is defined as before, with the addition that a terminal node that fulfills case (c) in the adapted version of Prop. 1 is colored ?.*

**Theorem 9.** *The resulting abstract product graph $G_\|$ is a closed subgraph of the game graph over $\hat{M}_1 \| \hat{M}_2$. In addition, the initial coloring function is correct and defined over all the terminal nodes in the subgraph.*

Along with Thm. 3, this implies that $G_{||}$ can be colored correctly (w.r.t. the model checking of $\varphi$ on $\hat{M}_1||\hat{M}_2$) using the 3-valued algorithm. If the initial node is colored by a definite color, then by Thm. 7 the result holds in $M_1||M_2$ as well and we are done.

**Compositional Refinement.** Since an abstraction is used, the result of the model checking can be $\perp$, in which case the coloring of [18,19] returns a failure cause that needs to be eliminated. The failure cause is either a literal whose value in a certain state is $\perp$, or a may transition of the underlying model which is not a must transition.

In our setting, the refinement step is done compositionally: If the failure cause is a literal $l$ whose value in the failure state of $\hat{M}_1||\hat{M}_2$ is $\perp$, then $l$ has to be a local literal of one of the components. This is because the abstraction is appropriate w.r.t. $AP_1 \cap AP_2$, which implies that no indefinite values for the joint atomic propositions occur in $\hat{M}_1||\hat{M}_2$. Thus, refinement need only be applied on the corresponding component.

Otherwise, the failure cause is a may transition (which is not a must transition) of $\hat{M}_1||\hat{M}_2$ that needs to be refined in order to result in a must transition or no transition at all. Let $((\hat{s}_1, \hat{s}_2), (\hat{s}'_1, \hat{s}'_2))$ be this may transition. Then it results from may transitions $(\hat{s}_1, \hat{s}'_1)$ and $(\hat{s}_2, \hat{s}'_2)$ of $\hat{M}_1$ and $\hat{M}_2$ resp., such that at least one of them is not a must transition. In order to refine $((\hat{s}_1, \hat{s}_2), (\hat{s}'_1, \hat{s}'_2))$, one needs to refine the individual may transitions in each component separately. If both of them are not must transitions, then refinement should be applied in each component. This is because a must transition in the composition results from must transitions in *both* components. Otherwise, refinement should only be applied in the component where it is not a must transition.

In each component where refinement is necessary, the refinement can be done as in [26,18,19]. Moreover, in each component we adopt the incremental approach of [26,18,19] and avoid unnecessary refinement. In this approach, only nodes with indefinite colors are refined. In our setting, this corresponds to the ?-subgraph of each component. The result is the following compositional abstraction-refinement loop.

```
Step 0  For $i \in \{1, 2\}$, abstract $M_i$ into $\hat{M}_i$ appropriately w.r.t. $AP_1 \cap AP_2$ (e.g. as in Def.12).
Step 1  Model check each $\hat{M}_i\!\uparrow$ separately (for $i \in \{1, 2\}$):
          1. Construct the game graph $G_i$ for $\varphi$ and $\hat{M}_i\!\uparrow$.
          2. Apply the 3-valued coloring on $G_i$. Let $\chi_i$ be the resulting coloring function.
        If $\chi_1(n_1^0)$ or $\chi_2(n_2^0)$ is definite, return the corresp.model checking result for $M_1||M_2$.
Step 2  Consider the composition $\hat{M}_1||\hat{M}_2$:
          1. Construct the ?-subgraphs for $G_1$ and $G_2$.
          2. Construct the (abstract) product graph $G_{||}$ of the ?-subgraphs.
          3. Apply the 3-valued coloring on $G_{||}$ (with the initial coloring function).
        If $\chi_{||}(n_{||}^0)$ is definite, return the corresp.model checking result for $M_1||M_2$.
Step 3  Refine: Consider the failure cause returned by the coloring of $G_{||}$ (where $\chi_{||}(n_{||}^0) = ?$).
        If it is $l \in Lit_i$ then refine $\hat{M}_i$; Else let it be the may transition $((\hat{s}_1, \hat{s}_2), (\hat{s}'_1, \hat{s}'_2))$. Then:
          1. If $(\hat{s}_1, \hat{s}'_1)$ is not a must transition in $\hat{M}_1$, refine $\hat{M}_1$.
          2. If $(\hat{s}_2, \hat{s}'_2)$ is not a must transition in $\hat{M}_2$, refine $\hat{M}_2$.
        Refine the ?-subgraphs of $G_1$ and $G_2$ accordingly (as in the incremental approach);
        Go to Step 1(2) with the refined subgraphs.
```

Note that the must transitions of each abstract component are only used when $G_{\|}$ is constructed. Thus, their computation can be deferred to step 2 and be limited to must transitions that are needed during model checking. Hyper-transitions can also be used, e.g. with the algorithm of [27].

Using the compositional abstraction-refinement starting from the most coarse abstraction w.r.t. $AP_1 \cap AP_2$ of one or both of the components results in the asymmetric, resp. symmetric, approach described in Section 5.1.

**Theorem 10.** *For finite concrete components, iterating the compositional abstraction-refinement process is guaranteed to terminate with a definite answer.*

## References

1. Aziz, V.A., Shiple, T.R., Sangiovanni-vincentelli, A.L.: Formula-dependent equivalence for compositional CTL model checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, Springer, Heidelberg (1994)
2. Alur, R., Cerny, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for Java classes. In: POPL (2005)
3. Alur, R., de Alfaro, L., Henzinger, T.A., Mang, F.Y.C.: Automating modular verification. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, Springer, Heidelberg (1999)
4. Alur, R., Henzinger, T., Kupferman, O.: Alternating-time temporal logic. In: FOCS (1997)
5. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, Springer, Heidelberg (2005)
6. Ball, T., Kupferman, O.: An abstraction-refinement framework for multi-agent systems. In: LICS (2006)
7. Barringer, H., Giannakopoulou, D., Pasareanu, C.: Proof rules for automated compositional verification through learning. In: SAVCBS (2003)
8. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, Springer, Heidelberg (1999)
9. Chaki, S., Clarke, E., Grumberg, O., Ouaknine, J., Sharygina, N., Touili, T., Veith, H.: State/event software verification for branching-time specifications. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, Springer, Heidelberg (2005)
10. Chaki, S., Clarke, E.M., Sinha, N., Thati, P.: Automated assume-guarantee reasoning for simulation conformance. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, Springer, Heidelberg (2005)
11. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
12. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) ETAPS 2003 and TACAS 2003. LNCS, vol. 2619, Springer, Heidelberg (2003)
13. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM Transactions on Programming Languages and Systems (TOPLAS), 19(2) (1997)
14. de Alfaro, L., Godefroid, P., Jagadeesan, R.: Three-valued abstractions of games: Uncertainty, but with precision. In: LICS (2004)
15. de Alfaro, L., Henzinger, T.A., Mang, F.Y.C.: Detecting errors before reaching them. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, Springer, Heidelberg (2000)

16. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Assumption generation for software component verification. In: ASE (2002)
17. Godefroid, P., Jagadeesan, R.: Automatic abstraction using generalized model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, Springer, Heidelberg (2002)
18. Grumberg, O., Lange, M., Leucker, M., Shoham, S.: Don't know in the $\mu$-calculus. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, Springer, Heidelberg (2005)
19. Grumberg, O., Lange, M., Leucker, M., Shoham, S.: When not losing is better than winning: Abstraction and refinement for the full $\mu$-calculus. Information and Compuatation (2007) doi: 10.1016/j.ic.2006.10.009
20. Grumberg, O., Long, D.: Model checking and modular verification. TOPLAS, 16(3) (1994)
21. Huth, M., Jagadeesan, R., Schmidt, D.: Modal transition systems: A foundation for three-valued program analysis. In: Sands, D. (ed.) ESOP 2001 and ETAPS 2001. LNCS, vol. 2028, Springer, Heidelberg (2001)
22. Jones, C.: Specification and design of (parallel) programs. In: IFIP (1983)
23. Kozen, D.: Results on the propositional $\mu$-calculus. TCS, 27 (1983)
24. Li, H.C., Krishnamurthi, S., Fisler, K.: Modular verification of open features using three-valued model checking. Autom. Softw. Eng., 12(3) (2005)
25. Pnueli, A.: In transition for global to modular temporal reasoning about programs. In: Logics and Models of Concurrent Systems, vol. 13 (1984)
26. Shoham, S., Grumberg, O.: A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, Springer, Heidelberg (2003) (to appear in TOCL)
27. Shoham, S., Grumberg, O.: 3-valued abstraction: More precision at less cost. In: LICS (2006)

# Formalised Inductive Reasoning in the Logic of Bunched Implications

James Brotherston[*]

Dept. of Computing, Imperial College London

**Abstract.** We present a framework for inductive definitions in the logic of bunched implications, BI, and formulate two sequent calculus proof systems for inductive reasoning in this framework. The first proof system adopts a traditional approach to inductive proof, extending the usual sequent calculus for predicate BI with explicit induction rules for the inductively defined predicates. The second system allows an alternative mode of reasoning with inductive definitions by *cyclic proof*. In this system, the induction rules are replaced by simple case-split rules, and the proof structures are cyclic graphs formed by identifying some sequent occurrences in a derivation tree. Because such proof structures are not sound in general, we demand that cyclic proofs must additionally satisfy a *global trace condition* that ensures soundness. We illustrate our inductive definition framework and proof systems with simple examples which indicate that, in our setting, cyclic proof may enjoy certain advantages over the traditional induction approach.

## 1 Introduction

The mechanised verification of properties of computer programs — for example, properties expressing safety, liveness, or correctness — is an important and very challenging problem currently attracting considerable interest in the computer science research community. A source of inconvenience, though, is the tendency of real-life computer programs to be written in low-level languages employing pointer arithmetic and similar operations that directly alter data stored in shared mutable structures, such as the heap. Because the (potentially dangerous) effects of these operations are hard to analyse, programs written using such languages have so far proven very much less amenable to formal reasoning than those written in, e.g., high-level functional programming languages, which are typically more well-behaved from a mathematical standpoint. The logic of *bunched implications* (BI), formulated by O'Hearn and Pym [19], addresses this problem by offering a convenient formalism for expressing properties of programs that access and modify some shared resource [16]. In this paper, we extend BI with a framework for *inductive definitions*, and formulate sequent calculus proof systems for formal reasoning in this extension.

[*] This research is supported by EPSRC grant EP/E002536/1. My thanks also to Richard Bornat, Cristiano Calcagno, Peter O'Hearn, David Pym, and Hongseok Yang for helpful discussions, references and advice.

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 87–103, 2007.

Inductive definitions are an important and well-established tool for representing many structures commonly used in the specification of computer programs, such as linked lists and binary trees. For any inductively defined structure, there are naturally associated inductive proof principles allowing us to reason about the structure by exploiting the recursion in its definition. Most often, these principles are encoded as inference rules or axioms in the native reasoning framework; but one can also reason with inductive definitions via a natural mode of *cyclic proof* [8,9]. In contrast to the usual finite tree proofs, cyclic proofs are regular infinite trees — represented as a finite (cyclic) graph — satisfying a global condition ensuring soundness. For inductively defined relations, this soundness condition is manifested as a generalisation of the principle of infinite descent *à la* Fermat [10].

By considering particular models of BI, one can obtain logics suitable for carrying out verification in specific programming languages. One very successful such logic is the *separation logic* of O'Hearn, Reynolds and Yang, which is suitable for reasoning about C-like languages [21]. Separation logic has to date been used in the verification of several non-trivial programs involving pointer arithmetic, including (but not limited to) a copying garbage collector [6], a DAG duplication program [7] and the Schorr-Waite graph marking algorithm [25]. It has also fruitfully been employed in local shape analysis [13,14], program termination analysis [4], and automated program verification (see e.g. [3,2]).

However, as has been noted [5,22], static analysis in separation logic (and other analysis based upon BI) has so far typically relied upon *ad hoc* extensions of the core logic by the particular inductive definitions needed for the development. It is thus of clear interest to develop a formal extension of BI in which one can define and reason about general inductive structures (over some suitable class of definitions). We provide one such extension, which could form a basis for theorem proving support to check logical implications in BI involving arbitrary inductive predicates (as needed, e.g., to accelerate fixed-point computations in shape analysis [13], or to check properties of predicates generated by inductive recursion synthesis [15] or used in automated verification [18]). Furthermore, our notion of *cyclic proof* for reasoning with inductive predicates appears to offer a new and potentially advantageous approach to certain static analysis applications (which we discuss later). In this paper, however, we confine ourselves to providing the foundations necessary to develop such applications.

In Section 2, we extend first-order predicate BI with a framework for (possibly mutual) inductively defined relations, based on simple "productions" in the style of Martin-Löf [17], in which the multiplicative connectives of BI may occur in the premises of definitions. This framework, though relatively simple, appears nonetheless powerful enough to express the inductive definitions that have arisen in practice in existing applications of separation logic to program verification. In Section 3 we extend the usual Gentzen-style proof system for BI to obtain a proof system supporting induction in the extended logic $\mathrm{BI}_{\mathrm{ID}}$ by adding left- and right-introduction rules for atomic formulas involving the inductive predicates of the theory. Following the approach taken in [8,9,10], the right introduction rules for an inductive predicate $P$ are merely sequent versions of the productions defining $P$, while the left-introduction rule for $P$ embodies the natural principle of rule induction over the definition of $P$. However, there is also a natural notion of cyclic proof for the logic, for which we introduce a second proof system in Section 4. In this system, the induction rules of the first system are replaced by simple *case-split* rules. *Pre-proofs* in the system are “unfinished” derivation trees in which every node to which no proof rule has been applied is identified with a syntactically identical interior node; pre-proofs can thus straightforwardly be understood as cyclic graphs. In general, pre-proofs are not sound, so to ensure soundness we impose a *global trace condition* stipulating, essentially, that for each infinite path in the pre-proof, some inductive definition is unfolded infinitely often along the path. By appealing to the well-foundedness of our inductive definitions, all such paths can be disregarded, whereby the remaining portion of proof is finite and hence sound for standard reasons. Finally, in Section 5, we identify the main directions for future work.

## 2 First-Order Predicate BI with Inductive Definitions

In this section we give the syntax and semantics of our logic, $\mathrm{BI_{ID}}$, obtained by extending first-order predicate BI à la Biering *et al* [5] with a framework for (possibly mutual) inductive definitions.

A brief comment on some of our mathematical and notational conventions is in order. We often use vector notation to abbreviate sequences, e.g. $\mathbf{x}$ for $(x_1, \ldots, x_n)$; for any $n \in \mathbb{N}$ and $i \leq n$ we define the $i$th projection function $\pi_i^n$ on $n$-tuples of sets by $\pi_i^n(X_1, \ldots, X_n) = X_i$; for any $n \in \mathbb{N}$ we extend set union, intersection and inclusion to $n$-tuples of sets by their corresponding pointwise definitions; and we write $\mathrm{Pow}(X)$ for the powerset of a set $X$.

Our languages are the standard (countable) first-order languages — containing arbitrarily many constant, function, and predicate symbols — except that we designate finitely many of the predicate symbols as *inductive.* A predicate symbol that is not inductive is called *ordinary.* For the rest of this paper, we shall consider a fixed language $\Sigma$ containing exactly $n$ inductive predicates $P_1, \ldots, P_n$, and use $Q_1, Q_2, \ldots$ for ordinary predicates. We also assume the existence of a denumerably infinite set $\mathcal{V}$ of variables, which is disjoint from $\Sigma$.

The elements of $\Sigma$ are interpreted by a structure, as in first-order logic, with the difference here that our structures include a notion of a set of possible resource states or “worlds”, given by a partial commutative monoid. The interpretation of predicates is parameterised by the elements of this monoid: in other words, the set of (tuples of) objects in the domain of which a given predicate is true depends on the current resource state. (However, the interpretations of the constant and function symbols are resource-independent).

**Definition 2.1 (BI-structure).** A BI-*structure* for $\Sigma$ is a tuple:

$$M = (D, \langle R, \circ, e \rangle, \mathbf{c}^M, \mathbf{f}^M, \mathbf{Q}^M, \mathbf{P}^M)$$

where $D$ is a set (called the *domain* of $M$), $\langle R, \circ, e\rangle$ is a partial commutative monoid and:

- $c^M \in D$ for each constant $\Sigma$-symbol $c \in \{\mathbf{c}\}$;
- $f^M : D^k \to D$ for each function $\Sigma$-symbol $f \in \{\mathbf{f}\}$ of arity $k$;
- $Q^M \subseteq R \times D^k$ for each ordinary predicate $\Sigma$-symbol $Q \in \{\mathbf{Q}\}$ of arity $k$;
- $P^M \subseteq R \times D^k$ for each inductive predicate $\Sigma$-symbol $P \in \{\mathbf{P}\}$ of arity $k$;

If $\mathbf{X}$ is an $n$-tuple of sets satisfying $\pi_i^n(\mathbf{X}) \subseteq R \times D^{k_i}$, where $k_i$ is the arity of $P_i$, for all $i \in \{1, \ldots, n\}$, then we write $M[\mathbf{P} \mapsto \mathbf{X}]$ to mean the structure defined as $M$ except that $\mathbf{P}^{M[\mathbf{P}\mapsto\mathbf{X}]} = \mathbf{X}$.

Our structures interpret the inductive predicate symbols of $\Sigma$ only for technical convenience: we shall only be interested later in those structures in which the interpretation of the inductive predicates coincides with the standard interpretation, which is determined by a fixed set of inductive definitions.

The *terms* of $\Sigma$ are defined as usual; we write $t[u/x]$ to denote the term obtained by substituting the term $u$ for all occurrences of the variable $x$ in the term $t$. We write $t(x_1, \ldots, x_n)$ for a term $t$ all of whose variables occur in $\{x_1, \ldots, x_n\}$, where $x_1, \ldots, x_n$ are distinct, and in such cases write $t(t_1, \ldots, t_n)$ to denote the term obtained by substituting $t_1, \ldots, t_n$ for $x_1, \ldots, x_n$ respectively in $t$. Also, if $M$ is a structure with domain $D$, then $t^M(x_1, \ldots, x_k) : D^k \to D$ is obtained by replacing every constant symbol $c$ by $c^M$ and every function symbol $f$ by $f^M$ in $t(x_1, \ldots, x_n)$.

The formulas of $\mathrm{BI}_{\mathrm{ID}}$ are just the standard formulas of predicate BI[1], given by the following grammar:

$$\begin{aligned} F ::= {} & \top \mid \bot \mid I \mid Q(t_1, \ldots, t_k)\ (k = \text{arity of } Q) \mid t_1 = t_2 \mid \\ & F \wedge F \mid F \vee F \mid F \to F \mid F * F \mid F \mathrel{-\!\!*} F \mid \exists x F \mid \forall x F \end{aligned}$$

where $Q$ ranges over all the predicate symbols of $\Sigma$ (both inductive and ordinary), $x$ ranges over $\mathcal{V}$ and $t_1, \ldots, t_k$ range over terms of $\Sigma$. We use the standard precedences on the logical connectives, with $*$ and $\mathrel{-\!\!*}$ having the same logical precedence as $\wedge$ and $\to$ respectively, and use parentheses to disambiguate where necessary. We write $\neg F$ to abbreviate the formula $F \to \bot$.

As in first-order logic, we interpret variables as elements of the domain $D$ of a BI-structure using environments $\rho : \mathcal{V} \to D$; we extend environments to all terms of $\Sigma$ in the usual way and write $\rho[x \mapsto d]$ for the environment defined exactly as $\rho$ except that $\rho[x \mapsto d](x) = d$. The formulas of $\mathrm{BI}_{\mathrm{ID}}$ are then interpreted by the following satisfaction (a.k.a. "forcing") relation:

**Definition 2.2 (Satisfaction relation for** BI**).** Let $M = (D, \langle R, \circ, e\rangle, \ldots)$ be a BI-structure for the language $\Sigma$, let $r \in R$ and let $\rho$ be an environment for $M$. We define the satisfaction relation $M, r \models_\rho F$ on formulas by:

[1] As in [5], our "predicate BI" is propositional BI extended with the usual additive quantifiers $\forall$ and $\exists$, as opposed to propositional BI extended with both additive and multiplicative versions of the quantifiers, as in e.g. [19].

$$\begin{array}{rcl}
M, r \models_\rho \top & \iff & \text{true} \\
M, r \models_\rho \bot & \iff & \text{false} \\
M, r \models_\rho I & \iff & r = e \\
M, r \models_\rho Q\mathbf{t} & \iff & Q^M(r, \rho(\mathbf{t})) \quad (Q \text{ ordinary or inductive}) \\
M, r \models_\rho t_1 = t_2 & \iff & \rho(t_1) = \rho(t_2) \\
M, r \models_\rho F_1 \wedge F_2 & \iff & M, r \models_\rho F_1 \text{ and } M, r \models_\rho F_2 \\
M, r \models_\rho F_1 \vee F_2 & \iff & M, r \models_\rho F_1 \text{ or } M, r \models_\rho F_2 \\
M, r \models_\rho F_1 \rightarrow F_2 & \iff & M, r \models_\rho F_1 \text{ implies } M, r \models_\rho F_2 \\
M, r \models_\rho F_1 * F_2 & \iff & r = r_1 \circ r_2 \text{ and } M, r_1 \models_\rho F_1 \text{ and } M, r_2 \models_\rho F_2 \\
 & & \text{for some } r_1, r_2 \in R \\
M, r \models_\rho F_1 \mathrel{-\!\!*} F_2 & \iff & \text{for all } r' \in R,\ M, r' \models_\rho F_1 \text{ and } r' \circ r \text{ defined} \\
 & & \text{implies } M, r' \circ r \models_\rho F_2 \\
M, r \models_\rho \forall x F & \iff & M, r \models_{\rho[x \mapsto d]} F \text{ for all } d \in D \\
M, r \models_\rho \exists x F & \iff & M, r \models_{\rho[x \mapsto d]} F \text{ for some } d \in D
\end{array}$$

(Informally, $M, r \models_\rho F$ means: "the formula $F$ is true in $M$ in the resource state $r$ and under the environment $\rho$").

We now give our schema for (possibly mutual) inductive definitions, which extends the framework used in [8,9,10] and, like that framework, is based on Martin-Löf's "productions" [17]. Our schema allows the multiplicative connectives of BI to occur in the premises of definitional clauses:

**Definition 2.3 (Inductive definition set).** An *inductive definition set* for $\Sigma$ is a set of *productions*, which are rules of the form:

$$\frac{C(\mathbf{x})}{P_i\mathbf{t}(\mathbf{x})} \quad i \in \{1, \ldots, n\}$$

where $C(\mathbf{x})$ is an *inductive clause* given by the following grammar:

$$\begin{aligned} C(\mathbf{x}) ::= \ & \top \mid I \mid Q\mathbf{t}(\mathbf{x}) \mid P_j\mathbf{t}(\mathbf{x}) \ (j \in \{1, \ldots, n\}) \mid t_1(\mathbf{x}) = t_2(\mathbf{x}) \mid \\ & C(\mathbf{x}) \wedge C(\mathbf{x}) \mid C(\mathbf{x}) * C(\mathbf{x}) \mid \hat{F}(\mathbf{x}) \rightarrow C(\mathbf{x}) \mid \hat{F}(\mathbf{x}) \mathrel{-\!\!*} C(\mathbf{x}) \mid \forall x C(\mathbf{x}) \end{aligned}$$

where $Q$ ranges over the ordinary predicate symbols of $\Sigma$ and $\hat{F}(\mathbf{x})$ ranges over all formulas of BI in which no inductive predicate symbols occur and whose free variables are contained in $\{\mathbf{x}\}$.

The productions whose conclusions feature an inductive predicate $P$ should be read as disjunctive clauses of the definition of $P$, whose free variables are implicitly existentially quantified. For some readers the following, equivalent notation for definitions may be more familiar:

$$P\mathbf{y} =_{def} (\exists \mathbf{x_1}.\, \mathbf{y} = \mathbf{t_1}(\mathbf{x_1}) \wedge C_1(\mathbf{x_1})) \vee \ldots \vee (\exists \mathbf{x_k}.\, \mathbf{y} = \mathbf{t_k}(\mathbf{x_k})) \wedge C_k(\mathbf{x_k}))$$

where $\{\mathbf{y}\} \cap \{\mathbf{x_1}, \ldots, \mathbf{x_k}\} = \emptyset$ and $C_1(\mathbf{x_1}), \ldots, C_k(\mathbf{x_k})$ are inductive clauses. It is trivial to convert from either form to the other.

As usual, the standard interpretation of the inductive predicate symbols of $\Sigma$ is obtained by taking the least fixed point of a monotone operator constructed from the definition set $\Phi$:

**Definition 2.4 (Definition set operator).** Let $M = (D, \langle R, \circ, e\rangle, \ldots)$ be a BI-structure for $\Sigma$, let $\Phi$ be an inductive definition set for $\Sigma$ and, for each $i \in \{1, \ldots, n\}$, let $k_i$ be the arity of the inductive predicate symbol $P_i$. We partition $\Phi$ into disjoint subsets $\Phi_1, \ldots, \Phi_n \subseteq \Phi$ by defining each $\Phi_i$ to be the set of productions in $\Phi$ in whose conclusion $P_i$ occurs. We then index each definition set $\Phi_i$ by $j$, with $j \in \{1, \ldots, |\Phi_i|\}$, and from each production $\Phi_{i,j} \in \Phi_i$, say:

$$\frac{C(\mathbf{x})}{P_i\mathbf{t}(\mathbf{x})}$$

we obtain a corresponding $n$-ary function $\varphi_{i,j} : (\mathrm{Pow}(R \times D^{k_1}) \times \ldots \times \mathrm{Pow}(R \times D^{k_n})) \to \mathrm{Pow}(R \times D^{k_i})$ as follows:

$$\varphi_{i,j}(\mathbf{X}) = \{(r, \mathbf{t}^M(\mathbf{d})) \mid M[\mathbf{P} \mapsto \mathbf{X}], r \models_{\rho[\mathbf{x}\mapsto\mathbf{d}]} C(\mathbf{x})\}$$

(Note that any variables occurring in the right hand side but not the left hand side of the set expression in the definition of $\varphi_{i,j}$ above are, implicitly, existentially quantified over the entire right hand side of the expression.) Then the *definition set operator* for $\Phi$ is the operator $\varphi_\Phi$, with domain and codomain $\mathrm{Pow}(R \times D^{k_1}) \times \ldots \times \mathrm{Pow}(R \times D^{k_n})$, defined by:

$$\varphi_\Phi(\mathbf{X}) = (\bigcup_j \varphi_{1,j}(\mathbf{X}), \ldots, \bigcup_j \varphi_{n,j}(\mathbf{X}))$$

**Proposition 2.5.** *The operator $\varphi_\Phi$ is monotone (with respect to $\subseteq$).*

*Proof.* (Sketch) Assuming that $\mathbf{X} \subseteq \mathbf{Y}$, where $\mathbf{X}$ and $\mathbf{Y}$ are $n$-tuples of sets of the appropriate type, one can prove by structural induction on $C(\mathbf{x})$ that $M[\mathbf{P} \mapsto \mathbf{X}], r \models_{\rho[\mathbf{x}\mapsto\mathbf{d}]} C(\mathbf{x})$ implies $M[\mathbf{P} \mapsto \mathbf{Y}], r \models_{\rho[\mathbf{x}\mapsto\mathbf{d}]} C(\mathbf{x})$. It follows that $\varphi_{i,j}(\mathbf{X}) \subseteq \varphi_{i,j}(\mathbf{Y})$ for any $i$ and $j$, and thus $\varphi_\Phi(\mathbf{X}) \subseteq \varphi_\Phi(\mathbf{Y})$ as required. □

*Example 2.6.* Let $\Phi_N$ be the inductive definition set consisting of the following productions for a unary inductive predicate $N$:

$$\frac{\top}{N0} \qquad \frac{Nx}{Nsx}$$

Then the definition set operator for $\Phi_N$ is defined by:

$$\varphi_{\Phi_N}(X) = \{(r, 0^M) \mid r \in R\} \cup \{(r, s^M d) \mid (r, d) \in X\}$$

In structures $M$ in which all "numerals" $(s^M)^k 0^M$ for $k \geq 0$ are distinct, the predicate $N$ corresponds to the property of being a natural number.

*Example 2.7.* Let $\mapsto$ be an ordinary, binary predicate symbol (written infix), and let $\Phi_{\mathtt{ls}}$ be the inductive definition set consisting of the following productions for a binary inductive predicate `ls`:

$$\frac{I}{\mathtt{ls}\,x\,x} \qquad \frac{x \mapsto x' * \mathtt{ls}\,x'\,y}{\mathtt{ls}\,x\,y}$$

Then the definition set operator for $\Phi_{\texttt{ls}}$ is defined by:

$$\begin{aligned}\varphi_{\Phi_{\texttt{ls}}}(X) = \{&(e,(d,d)) \mid d \in D\} \\ &\cup \{(r_1 \circ r_2,(d,d')) \mid (r_1,(d,d'')) \in \mapsto^M \text{ and } (r_2,(d'',d')) \in X\}\end{aligned}$$

where $d''$ in the second set comprehension is, implicitly, existentially quantified. In separation logic, where the resource states are heaps and $x \mapsto y$ is true of a heap $h$ if $h$ is a single-celled heap in which $x$ is a pointer to $y$, the predicate `ls` is used to represent (possibly cyclic) segments of singly-linked lists, so that $\texttt{ls}\,x\,y$ is true of a heap $h$ if $h$ represents a linked list whose first element is pointed to by $x$ and whose last element contains the pointer $y$.

It is a standard result for inductive definitions that the least $n$-tuple of sets closed under the productions in $\Phi$ is the least prefixed point of the operator $\varphi_\Phi$ (see e.g. [1]), and that this least prefixed point can be approached in iterative *approximant* stages, as follows:

**Definition 2.8 (Approximants).** Let $\Phi$ be an inductive definition set for $\Sigma$, and define a chain of ordinal-indexed sets $(\varphi_\Phi^\alpha)_{\alpha \geq 0}$ by transfinite induction: $\varphi_\Phi^\alpha = \bigcup_{\beta<\alpha} \varphi_\Phi(\varphi_\Phi^\beta)$ (note that this implies $\varphi_\Phi^0 = (\emptyset, \dots, \emptyset)$). Then for each $i \in \{1, \dots, n\}$, the set $P_i^\alpha = \pi_i^n(\varphi_\Phi^\alpha)$ is called the $\alpha^{th}$ *approximant of* $P_i$.

**Definition 2.9 (Standard model).** Let $\Phi$ be an inductive definition set for $\Sigma$. Then a BI-structure $M$ for $\Sigma$ is said to be a *standard model* for $(\Sigma, \Phi)$ if $P_i^M = \bigcup_\alpha P_i^\alpha$ for all $i \in \{1, \dots, n\}$.

Definition 2.9 thus fixes within a BI-structure a standard interpretation of the inductive predicate symbols of $\Sigma$ that is uniquely determined by the other components of the structure.

**Proposition 2.10.** *For any inductive definition set $\Phi$* not *employing universal quantification, $\varphi_\Phi^\omega$ is a prefixed point of $\varphi_\Phi$ and thus in a standard model of $(\Sigma, \Phi)$ we have $P_i^M = P_i^\omega$ for all $i \in \{1, \dots, n\}$. If $\Phi$ does feature universal quantification, the closure ordinal is $> \omega$ in general.*

*Proof.* (Sketch) One can show by structural induction on $C(\mathbf{x})$ that, if $C(\mathbf{x})$ contains no occurrences of $\forall$, then $M[\mathbf{P} \mapsto \varphi_\Phi^\omega], r \models_{\rho[\mathbf{x}\mapsto\mathbf{d}]} C(\mathbf{x})$ implies $M[\mathbf{P} \mapsto \varphi_\Phi^k], r \models_{\rho[\mathbf{x}\mapsto\mathbf{d}]} C(\mathbf{x})$ for some $k \in \mathbb{N}$. It follows that $\varphi_{i,j}(\varphi_\Phi^\omega) \subseteq \bigcup_{k\in\mathbb{N}} \varphi_{i,j}(\varphi_\Phi^k)$ for any $i$ and $j$, and thus that $\varphi_\Phi(\varphi_\Phi^\omega) \subseteq \varphi_\Phi^\omega$ as required.

For the second part of the proposition, consider a BI-structure $M$ with domain $\mathbb{N}$ and in which the Peano axioms hold, and the inductive definition set $\Phi$ consisting of the productions for $N$ in Example 2.6 together with a production with premise $\forall x N x$ and conclusion $P0$ (where $P$ is a unary inductive predicate symbol). Then one can easily verify that the least prefixed point of $\varphi_\Phi$ is $\varphi_\Phi^{\omega+1}$. □

## 3 A Proof System for Induction in $\mathrm{BI_{ID}}$

In this section we give a Gentzen-style proof system suitable for formalising traditional proof by induction in our logic $\mathrm{BI_{ID}}$. We fix an inductive definition set $\Phi$ for $\Sigma$, partitioned into $\Phi_1, \ldots, \Phi_n$ as in Defn. 2.4. Our starting point will be the standard sequent calculus for BI (cf. [20]). We write *sequents* of the form $\Gamma \vdash F$, where $F$ is a formula and $\Gamma$ is a *bunch*, given by the following definition:

**Definition 3.1 (Bunch).** A *bunch* is a tree whose leaves are labelled by formulas of $\mathrm{BI_{ID}}$ and whose internal nodes are labelled by ';' or ',' (denoting respectively additive and multiplicative combination).

As our sequents have at most one formula occurring on the right hand side, our proof system is intuitionistic. This is not for ideological reasons but for technical convenience; the formulation of a classical (multiple-conclusion) sequent calculus for BI would necessitate the use of a multiplicative disjunction (for details see [20]).

We write $\Gamma(\Delta)$ to mean that $\Gamma$ is a bunch of which $\Delta$ is a subtree (also called a "sub-bunch"), and write $\Gamma(\Delta')$ for the bunch obtained by replacing the considered instance of $\Delta$ by $\Delta'$ in $\Gamma(\Delta)$.

**Definition 3.2 (Coherent equivalence for bunches).** Define $\equiv$ to be the least relation on bunches satisfying:

1. commutative monoid equations for ';' and $\top$;
2. commutative monoid equations for ',' and $I$;
3. congruence: if $\Delta \equiv \Delta'$ then $\Gamma(\Delta) \equiv \Gamma(\Delta')$.

The usual sequent calculus rules for our version of predicate BI, plus rules for equality and an explicit substitution rule, are given in Figure 1. Our proof system, called $\mathrm{LBI_{ID}}$, is obtained from this system by adding rules for introducing atomic formulas of the form $P_i\mathbf{t}$, where $P_i$ is an inductive predicate symbol, on the left and right of sequents.

First, for each $i \in \{1, \ldots, n\}$ and each production $\Phi_{i,j} \in \Phi_i$, we obtain a right-introduction rule $(P_iR_j)$ for the predicate $P_i$ as follows:

$$\frac{C(\mathbf{x})}{P_i\mathbf{t}(\mathbf{x})} \quad \Longrightarrow \quad \frac{\Gamma \vdash C(\mathbf{u})}{\Gamma \vdash P_i\mathbf{t}(\mathbf{u})}\,(P_iR_j)$$

Before giving the rules for introducing inductive predicates on the left of sequents, we first give a formal definition of what it means for two inductive predicates to have a mutual definition in $\Phi$ (repeated from [8]):

**Definition 3.3 (Mutual dependency).** Define the binary relation *Prem* on the inductive predicate symbols $\{P_1, \ldots, P_n\}$ of $\Sigma$ as the least relation satisfying: $Prem(P_i, P_j)$ holds whenever $P_j$ occurs in the premise of some production in $\Phi_i$. Also define $Prem^*$ to be the reflexive-transitive closure of *Prem*. Then we say two predicate symbols $P$ and $Q$ are *mutually dependent* if both $Prem^*(P, Q)$ and $Prem^*(Q, P)$ hold.

**Structural rules:**

$$\frac{}{F \vdash F}\,(\text{Id}) \qquad \frac{\Gamma(\Delta) \vdash F}{\Gamma(\Delta;\Delta') \vdash F}\,(\text{Weak}) \qquad \frac{\Gamma(\Delta;\Delta) \vdash F}{\Gamma(\Delta) \vdash F}\,(\text{Contr})$$

$$\frac{\Gamma' \vdash F}{\Gamma \vdash F}\;\Gamma \equiv \Gamma'\;(\text{Equiv}) \qquad \frac{\Delta \vdash G \quad \Gamma(G) \vdash F}{\Gamma(\Delta) \vdash F}\,(\text{Cut}) \qquad \frac{\Gamma \vdash F}{\Gamma[\theta] \vdash F[\theta]}\,(\text{Subst})$$

**Propositional rules:**

$$\frac{}{\bot \vdash F}\,(\bot\text{L}) \qquad \frac{\Gamma(F_1) \vdash F \quad \Gamma(F_2) \vdash F}{\Gamma(F_1 \vee F_2) \vdash F}\,(\vee\text{L}) \qquad \frac{\Gamma(F_1;F_2) \vdash F}{\Gamma(F_1 \wedge F_2) \vdash F}\,(\wedge\text{L})$$

$$\frac{}{\vdash \top}\,(\top\text{R}) \qquad \frac{\Gamma \vdash F_i}{\Gamma \vdash F_1 \vee F_2}\; i \in \{1,2\}\;(\vee\text{R}) \qquad \frac{\Gamma \vdash F_1 \quad \Gamma \vdash F_2}{\Gamma \vdash F_1 \wedge F_2}\,(\wedge\text{R})$$

$$\frac{\Delta \vdash F_1 \quad \Gamma(F_2) \vdash F}{\Gamma(\Delta, F_1 \mathrel{-\!\!*} F_2) \vdash F}\,(\mathrel{-\!\!*}\text{L}) \qquad \frac{\Delta \vdash F_1 \quad \Gamma(\Delta;F_2) \vdash F}{\Gamma(\Delta;F_1 \to F_2) \vdash F}\,(\to\text{L}) \qquad \frac{\Gamma(F_1,F_2) \vdash F}{\Gamma(F_1 * F_2) \vdash F}\,(*\text{L})$$

$$\frac{\Gamma, F_1 \vdash F_2}{\Gamma \vdash F_1 \mathrel{-\!\!*} F_2}\,(\mathrel{-\!\!*}\text{R}) \qquad \frac{\Gamma; F_1 \vdash F_2}{\Gamma \vdash F_1 \to F_2}\,(\to\text{R}) \qquad \frac{\Gamma \vdash F_1 \quad \Delta \vdash F_2}{\Gamma, \Delta \vdash F_1 * F_2}\,(*\text{R})$$

**Quantifier rules:**

$$\frac{\Gamma(G[t/x]) \vdash F}{\Gamma(\forall x G) \vdash F}\,(\forall\text{L}) \qquad \frac{\Gamma \vdash F}{\Gamma \vdash \forall x F}\; x \notin FV(\Gamma)\;(\forall\text{R})$$

$$\frac{\Gamma(G) \vdash F}{\Gamma(\exists x G) \vdash F}\; x \notin FV(\Gamma \cup \{F\})\;(\exists\text{L}) \qquad \frac{\Gamma \vdash F[t/x]}{\Gamma \vdash \exists x F}\,(\exists\text{R})$$

**Equality rules:**

$$\frac{}{\Gamma \vdash t = t}\,(=\text{R}) \qquad \frac{\Gamma(\top)[u/x,t/y] \vdash F[u/x,t/y]}{\Gamma(t = u)[t/x,u/y] \vdash F[t/x,u/y]}\,(=\text{L})$$

**Fig. 1.** Sequent calculus proof rules for predicate BI with equality

Now to obtain an instance of the induction rule for any inductive predicate $P_j$, we first associate with every inductive predicate $P_i$ a tuple $\mathbf{z_i}$ of $k_i$ distinct variables (called *induction variables*), where $k_i$ is the arity of $P_i$. Furthermore, we associate to every predicate $P_i$ that is mutually dependent with $P_j$ a formula (called an *induction hypothesis*) $H_i$, possibly containing some of the induction variables. Next, define the formula $G_i$ for each $i \in \{1, \ldots, n\}$ by: $G_i = H_i$ if $P_i$ and $P_j$ are mutually dependent, and $G_i = P_i\mathbf{z_i}$ otherwise. For convenience, we shall write $G_i\mathbf{t}$ for $G_i[\mathbf{t}/\mathbf{z_i}]$, where $\mathbf{t}$ is a tuple of $k_i$ terms. Then an instance of the induction rule (Ind $P_j$) for $P_j$ has the following schema:

$$\frac{\text{minor premises} \qquad \Gamma(\Delta; H_j\mathbf{t}) \vdash F}{\Gamma(\Delta; P_j\mathbf{t}) \vdash F}\ (\text{Ind } P_j)$$

where the premise $\Gamma(\Delta; H_j\mathbf{t}) \vdash F$ is called the *major premise* of the rule, and for each predicate $P_i$ that is mutually dependent with $P_j$, we obtain a *minor premise* from every production in $\Phi_i$ as follows:

$$\frac{C(\mathbf{x})}{P_i\mathbf{t}(\mathbf{x})} \qquad \Longrightarrow \qquad \Delta; C_H(\mathbf{x}) \vdash H_i\mathbf{t}(\mathbf{x}) \quad (\forall x \in \mathbf{x}.\, x \notin FV(\Delta))$$

where $C_H(\mathbf{x})$ is the formula obtained by replacing every formula of the form $P_k\mathbf{t}(\mathbf{x})$ (for $P_k$ an inductive predicate) by $G_k\mathbf{t}(\mathbf{x})$ in the inductive clause $C(\mathbf{x})$.

*Example 3.4.* The induction rule for the predicate $N$ from Example 2.6 is:

$$\frac{\Delta \vdash H0 \quad \Delta; Hx \vdash Hsx \quad \Gamma(\Delta; Ht) \vdash F}{\Gamma(\Delta; Nt) \vdash F}\ (\text{Ind } N)$$

where $H$ is the induction hypothesis associated with $N$ and $x$ is suitably fresh.

*Example 3.5.* The induction rule for the predicate `ls` from Example 2.7 is:

$$\frac{\Delta; I \vdash Hxx \quad \Delta; x \mapsto x' * Hx'y \vdash Hxy \quad \Gamma(\Delta; Htu) \vdash F}{\Gamma(\Delta; \mathtt{ls}\, t\, u) \vdash F}\ (\text{Ind } \mathtt{ls})$$

where $H$ is the induction hypothesis associated with `ls` and $x, x', y$ are fresh.

**Definition 3.6 (Validity).** Let $M$ be a standard model for $(\Sigma, \Phi)$. Then a sequent $\Gamma \vdash F$ is said to be *true in* $M$ if $M, r \models_\rho \phi_\Gamma$ implies $M, r \models_\rho F$ for all environments $\rho$ and resource states $r$, where $\phi_\Gamma$ is the formula obtained by replacing every occurrence of ';' by $\wedge$ and every occurrence of ',' by $*$ in the bunch $\Gamma$. $\Gamma \vdash F$ is said to be *valid* if it is true in all standard models.

By a *derivation tree*, we mean a finite tree of sequents in which each parent sequent is obtained as the conclusion of an inference rule with its children as premises. We distinguish between "leaves" and "buds" in the tree. By a *leaf* we mean an axiom, i.e., the conclusion of a 0-premise inference rule. By a *bud* we mean any sequent occurrence in the tree that is not the conclusion of a proof rule. An $\mathrm{LBI}_{\mathrm{ID}}$ *proof* is then, as usual, a finite derivation tree constructed according to the proof rules that contains no buds. The following proposition is a straightforward consequence of the local soundness of our proof rules.

**Proposition 3.7 (Soundness of $\mathbf{LBI_{ID}}$).** *If there is an $LBI_{ID}$ proof of $\Gamma \vdash \Delta$ then $\Gamma \vdash \Delta$ is valid.*

*Example 3.8.* We give an $\mathrm{LBI}_{\mathrm{ID}}$ proof that the predicate $N$ from Example 2.6 admits multiplicative weakening, i.e. that $F, Nx \vdash Nx$:

$$\dfrac{\dfrac{\dfrac{}{F \vdash N0}\,(NR_1)}{\vdash F \mathrel{-\!\!*} N0}\,(\mathrel{-\!\!*}\mathrm{R}) \quad \dfrac{\dfrac{\dfrac{\dfrac{}{F \vdash F}\,(\mathrm{Id}) \quad \dfrac{}{Ny \vdash Ny}\,(\mathrm{Id})}{F, F \mathrel{-\!\!*} Ny \vdash Ny}\,(\mathrel{-\!\!*}\mathrm{L})}{F, F \mathrel{-\!\!*} Ny \vdash Nsy}\,(NR_2)}{F \mathrel{-\!\!*} Ny \vdash F \mathrel{-\!\!*} Nsy}\,(\mathrel{-\!\!*}\mathrm{R}) \quad \dfrac{\dfrac{}{F \vdash F}\,(\mathrm{Id}) \quad \dfrac{}{Nx \vdash Nx}\,(\mathrm{Id})}{F, F \mathrel{-\!\!*} Nx \vdash Nx}\,(\mathrel{-\!\!*}\mathrm{L})}{F, Nx \vdash Nx}\,(\mathrm{Ind}\ N)$$

Note that in the application of (Ind $N$) in this proof we associate the induction variable $z$ and the induction hypothesis $F \mathrel{-\!\!*} Nz$ with the inductive predicate $N$. We remark that one can easily see that this example demonstrates the need for generalisation of induction hypotheses in this setting (at least for cut-free proofs): it is clear that no subformula of the root sequent is sufficiently strong as an induction hypothesis to enable us to prove the major premise of the induction.

## 4 A Cyclic Proof System for $\mathrm{BI}_{\mathrm{ID}}$

We now define a second proof system $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ for $\mathrm{BI}_{\mathrm{ID}}$ which admits a notion of cyclic proof. *Pre-proofs* are finite derivation trees together with a function assigning to every bud in the tree a syntactically identical interior node (a *companion* for the bud), and thus can be viewed as cyclic graphs. Since pre-proofs are not sound in general, we impose a *global trace condition* on pre-proofs, corresponding to an infinite descent principle for our inductive definitions, to ensure soundness.

The proof rules of the system $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ are the rules of $\mathrm{LBI}_{\mathrm{ID}}$ described in Section 3, except that for each inductive predicate $P_j$ of $\Sigma$, the induction rule (Ind $P_j$) of $\mathrm{LBI}_{\mathrm{ID}}$ is replaced by the *case-split rule*:

$$\dfrac{\text{case distinctions}}{\Gamma(P_j\mathbf{u}) \vdash F}\,(\mathrm{Case}\ P_j)$$

where we obtain a case distinction from each production in $\Phi_j$ as follows:

$$\dfrac{C(\mathbf{x})}{P_j\mathbf{t}(\mathbf{x})} \qquad \Longrightarrow \qquad \Gamma(\mathbf{u} = \mathbf{t}(\mathbf{x}); C(\mathbf{x})) \vdash F \quad (\forall x \in \mathbf{x}.\, x \notin FV(\Gamma \cup \{F\}))$$

*Example 4.1.* The case-split rule for $N$ from Example 2.6 is:

$$\dfrac{\Gamma(t = 0; \top) \vdash F \qquad \Gamma(t = sx; Nx) \vdash F}{\Gamma(Nt) \vdash F}\,(\mathrm{Case}\ N)$$

*Example 4.2.* The case-split rule for `ls` from Example 2.7 (modulo applying the equality rule to eliminate some generated equalities) is:

$$\frac{\Gamma(t = u; I) \vdash F \qquad \Gamma(t \mapsto x * \mathtt{ls}\, x\, u) \vdash F}{\Gamma(\mathtt{ls}\, t\, u) \vdash F}\ (\text{Case } \mathtt{ls})$$

**Definition 4.3 (Companion).** Let $B$ be a bud of a derivation tree $\mathcal{D}$. A non-bud sequent $C$ in $\mathcal{D}$ is said to be a *companion* for $B$ if $C = B$.

By assigning a companion to each bud node in a finite derivation tree, one obtains a finite representation of an associated (regular) infinite tree:

**Definition 4.4 ($\mathbf{CLBI}^{\omega}_{\mathbf{ID}}$ pre-proof).** A *$CLBI^{\omega}_{ID}$ pre-proof* of a sequent $\Gamma \vdash \Delta$ is a pair $\mathcal{P} = (\mathcal{D}, \mathcal{R})$, where $\mathcal{D}$ is a derivation tree constructed according to the proof rules of $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ given above and whose root is $\Gamma \vdash \Delta$, and $\mathcal{R}$ is a function assigning a companion to every bud of $\mathcal{D}$.

We consider $\mathcal{D}$ to have a directed edge from the conclusion of each rule instance to each of its premises, whence the *graph* of $\mathcal{P}$ is the directed graph $\mathcal{G}_{\mathcal{P}}$ obtained from $\mathcal{D}$ by identifying each bud node $B$ in $\mathcal{D}$ with its companion $\mathcal{R}(B)$.

We observe that the local soundness of our proof rules is not sufficient to guarantee that pre-proofs are sound, due to the (possible) cyclicity evident in their graph representations. In order to give a criterion for soundness, we formulate the notion of a *trace* following a path in a pre-proof graph, similar to that used in [8,9,10,24] but more complex due to our richer induction schema and use of bunches in sequents:

**Definition 4.5 (Trace).** Let $\mathcal{P}$ be a $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ pre-proof and let $(\Gamma_i \vdash F_i)_{i \geq 0}$ be a path in $\mathcal{G}_{\mathcal{P}}$. A *trace following* $(\Gamma_i \vdash F_i)_{i \geq 0}$ is a sequence $(\tau_i)_{i \geq 0}$ such that, for all $i$, $\tau_i$ is a leaf of $\Gamma_i$ (we write $F_{\tau_i}$ to mean the formula labelling $\tau_i$ in $\Gamma_i$.) Furthermore, for each $i$, one of the following conditions must hold:

1. $\Gamma_i \vdash F_i$ is the conclusion of one of the following inferences, $\tau_i$ is the leaf of $\Gamma_i$ indicated by the underlined formula in the conclusion and $\tau_{i+1}$ is one of the leaves of $\Gamma_{i+1}$ indicated by the underlined formulas in the appropriate premise:

$$\frac{\Gamma(\underline{F_1}; \underline{F_2}) \vdash F}{\Gamma(\underline{F_1 \wedge F_2}) \vdash F}\ (\wedge\mathrm{L}) \quad \frac{\Gamma(\underline{F_1}, \underline{F_2}) \vdash F}{\Gamma(\underline{F_1 * F_2}) \vdash F}\ (*\mathrm{L}) \quad \frac{\ldots \Gamma(\mathbf{u} = \mathbf{t}(\mathbf{x}); \underline{C(\mathbf{x})}) \vdash F \ldots}{\Gamma(\underline{P_j \mathbf{u}}) \vdash F}\ (\text{Case } P_j)$$

$$\frac{\Delta \vdash F_1 \quad \Gamma(\underline{F_2}) \vdash F}{\Gamma(\Delta, \underline{F_1 \mathrel{-\!\!*} F_2}) \vdash F}\ (\mathrel{-\!\!*}\mathrm{L}) \quad \frac{\Delta \vdash F_1 \quad \Gamma(\Delta; \underline{F_2}) \vdash F}{\Gamma(\Delta; \underline{F_1 \to F_2}) \vdash F}\ (\to\mathrm{L}) \quad \frac{\Gamma(\underline{G[t/x]}) \vdash F}{\Gamma(\underline{\forall x G}) \vdash F}\ (\forall\mathrm{L})$$

   In the case where $\tau_i$ and $\tau_{i+1}$ are the leaves indicated by the underlined formulas in the displayed instance of (Case $P_j$) above, $i$ is said to be a *progress point* of the trace. An *infinitely progressing trace* is a trace having infinitely many progress points.

2. $\tau_{i+1}$ is the leaf in $\Gamma_{i+1}$ corresponding to $\tau_i$ in $\Gamma_i$, modulo any splitting of $\Gamma_i$ performed by the rule applied with conclusion $\Gamma_i \vdash F_i$. (Thus $F_{\tau_{i+1}} = F_{\tau_i}$, modulo any substitution performed by the rule.) E.g. if $\Gamma_i \vdash F_i$ is the conclusion of the inference:

$$\frac{\Delta \vdash F_1 \quad \Gamma(F_2) \vdash F}{\Gamma(\Delta, F_1 \mathrel{-\!\!*} F_2) \vdash F}\,(\mathrel{-\!\!*}\mathrm{L})$$

then, if $\Gamma_{i+1} \vdash F_{i+1}$ is the left hand premise, then $\tau_{i+1}$ and $\tau_i$ are the same leaf in $\Delta$ and, if $\Gamma_{i+1} \vdash F_{i+1}$ is the right hand premise, then $\tau_{i+1}$ and $\tau_i$ are the same leaf in $\Gamma(-)$.

Informally, a trace follows (a part of) the construction of an inductively defined predicate occurring in some part of the bunches occurring on a path in a pre-proof. These predicate constructions never become larger as we follow the trace along the path, and at progress points, they actually decrease. This property is encapsulated in the following lemma and motivates the subsequent definition of a *cyclic proof*:

**Lemma 4.6.** *Let $\mathcal{P}$ be a $CLBI^{\omega}_{ID}$ pre-proof of $\Gamma_0 \vdash F_0$, and let $M$ be a standard model such that $\Gamma_0 \vdash F_0$ is false in $M$ in the resource state $r_0$ and environment $\rho_0$ (say). Then there we can construct an infinite path $(\Gamma_i \vdash F_i)_{i\geq 0}$ in $\mathcal{G}_{\mathcal{P}}$ and infinite sequences $(r_i)_{i\geq 0}$ and $(\rho_i)_{i\geq 0}$ such that:*

1. *for all $i$, $\Gamma_i \vdash F_i$ is false in $M_i$ in the resource state $r_i$ and environment $\rho_i$;*
2. *if there is a trace $(\tau_i)_{i\geq n}$ following some tail $(\Gamma_i \vdash F_i)_{i\geq n}$ of $(\Gamma_i \vdash F_i)_{i\geq 0}$, then there exists a second sequence of resource states, $(r'_i)_{i\geq n}$, such that $M, r'_i \models_{\rho_i} F_{\tau_i}$ for all $i \geq n$ and the sequence $(\alpha_i)_{i\geq n}$ of ordinals defined by $\alpha_i =$ least $\alpha$ s.t. $M[\mathbf{P} \mapsto \varphi^{\alpha}_{\Phi}], r'_i \models_{\rho_i} F_{\tau_i}$, is non-increasing. Furthermore, if $j$ is a progress point of $(\tau_i)_{i\geq n}$ then $\alpha_{j+1} < \alpha_j$.*

*Proof.* (Sketch) To construct the required infinite sequences satisfying property 1 of the lemma just requires us to use the fact that the proof rules of $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ are locally sound (i.e., falsifiability of the conclusion of a rule instance implies falsifiability of one of its premises).

For part 2 of the lemma, we suppose there is a trace $(\tau_i)_{i\geq n}$ following the tail $(\Gamma_i \vdash F_i)_{i\geq n}$ of the constructed infinite path. Since $\Gamma_n \vdash F_n$ is false in $M$ under $\rho_n$ and $r_n$ by property 1, it follows that there is a suitable substate $r'_n$ of $r_n$ ("suitability" being given by a formal relation capturing the property that the relationship between $r'_n$ and $r_n$ reflects the position of $\tau_n$ in $\Gamma_n$) such that $M, r'_n \models_{\rho_n} F_{\tau_n}$, and thus there is a least ordinal $\alpha$ such that $M[\mathbf{P} \mapsto \varphi^{\alpha}_{\Phi}], r'_n \models_{\rho_n} F_{\tau_n}$. Now, given any edge $(\Gamma_k \vdash F_k, \Gamma_{k+1} \vdash F_{k+1})$ in the tail and a suitable substate $r'_k$ of $r_k$ satisfying $M, r'_k \models_{\rho_k} F_{\tau_k}$, one can find a suitable substate $r'_{k+1}$ of $r_{k+1}$ satisfying $M, r'_k \models_{\rho_k} F_{\tau_k}$. Moreover, we have:

$$\text{least } \alpha \text{ s.t. } M[\mathbf{P} \mapsto \varphi^{\alpha}_{\Phi}], r'_{k+1} \models_{\rho} F_{\tau_{k+1}} \leq \text{least } \alpha \text{ s.t. } M[\mathbf{P} \mapsto \varphi^{\alpha}_{\Phi}], r'_k \models_{\rho} F_{\tau_k}$$

Furthermore, if $k$ is a progress point of the trace, then this inequality holds strictly. This property, which can be proven by a lengthy case analysis on the rule with conclusion $\Gamma_k \vdash F_k$ enables us to construct the required sequences $(r'_i)_{i \geq n}$ and $(\alpha_i)_{i \geq n}$.

**Definition 4.7 (CLBI$^{\omega}_{\mathbf{ID}}$ proof).** A CLBI$^{\omega}_{\mathrm{ID}}$ pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ is a *CLBI$^{\omega}_{ID}$ proof* if, for every infinite path in $\mathcal{D}$, there is an infinitely progressing trace following some tail of the path.

**Proposition 4.8 (Soundness).** *If there is a CLBI$^{\omega}_{ID}$ proof of $\Gamma \vdash \Delta$ then $\Gamma \vdash \Delta$ is valid.*

*Proof.* If $\Gamma \vdash F$ has a CLBI$^{\omega}_{\mathrm{ID}}$ proof $\mathcal{P}$ but is false in some standard model $M$ then we can use property 1 of Lemma 4.6 to construct an infinite path $\pi$ in $\mathcal{G}_{\mathcal{P}}$ together with a sequence of environments and resource states that falsify each sequent along the path. Since $\mathcal{P}$ is a proof, there is an infinitely progressing trace following some tail of $\pi$. Thus we can invoke property 2 of Lemma 4.6 to create a monotonically decreasing chain of ordinals which, since the trace progresses infinitely often, must decrease infinitely often. This contradicts the well-foundedness of the ordinals, so $\Gamma \vdash F$ must indeed be valid. □

*Example 4.9.* The following is a CLBI$^{\omega}_{\mathrm{ID}}$ proof of the sequent $F, Nx \vdash Nx$ (recall we gave an LBI$_{\mathrm{ID}}$ proof in Example 3.8):

$$\dfrac{\dfrac{\dfrac{}{F \vdash N0}\,(NR_1)}{F, x = 0 \vdash Nx}\,(=\mathrm{L}) \qquad \dfrac{\dfrac{\dfrac{F, \underline{Nx} \vdash Nx \;\; (\dagger)}{F, \underline{Ny} \vdash Ny}\,(\mathrm{Subst})}{F, \underline{Ny} \vdash Nsy}\,(NR_2)}{F, (x = sy; \underline{Ny}) \vdash Nx}\,(=\mathrm{L})}{F, \underline{Nx} \vdash Nx \;\; (\dagger)}\,(\mathrm{Case}\ N)$$

We use $(\dagger)$ to indicate the pairing of a suitable companion with the only bud in this pre-proof. To see that it is indeed a CLBI$^{\omega}_{\mathrm{ID}}$ proof, observe that any infinite path $\pi$ in the pre-proof graph necessarily has a tail consisting of repetitions of the path from the companion to the bud in this pre-proof, and there is a progressing trace following this path, denoted by the underlined formulas (with a progress point at the displayed application of (Case $N$)). Thus by concatenating copies of this trace we can obtain an infinitely progressing trace on a tail of $\pi$ as required.

We remark that, unlike the situation for LBI$_{\mathrm{ID}}$ (cf. Example 3.8), we do not require generalisation in this proof, i.e., the invention of new formulas in the proof is not necessary.

*Example 4.10.* The following is a CLBI$^{\omega}_{\mathrm{ID}}$ pre-proof of $\mathtt{ls}\,x\,x', \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{ls}\,x\,y \vdash \mathtt{ls}\,x\,y}\,(\mathrm{Id})}{I, \mathtt{ls}\,x\,y \vdash \mathtt{ls}\,x\,y}\,(\equiv)}{(x' = x; I), \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y}\,(=\mathrm{L}) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{x \mapsto z \vdash x \mapsto z}\,(\mathrm{Id}) \quad \dfrac{(\dagger)\;\; \underline{\mathtt{ls}\,x\,x'}, \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y}{\underline{\mathtt{ls}\,z\,x'}, \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,z\,y}\,(\mathrm{Subst})}{x \mapsto z, \underline{\mathtt{ls}\,z\,x'}, \mathtt{ls}\,x'\,y \vdash x \mapsto z * \mathtt{ls}\,z\,y}\,({*}\mathrm{R})}{x \mapsto z, \underline{\mathtt{ls}\,z\,x'}, \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y}\,(\mathtt{ls}R_2)}{\underline{x \mapsto z * \mathtt{ls}\,z\,x'}, \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y}\,({*}\mathrm{L})}{(\dagger)\;\; \underline{\mathtt{ls}\,x\,x'}, \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y}\,(\mathrm{Case}\ \mathtt{ls})$$

The pairing of a suitable companion with the only bud in this pre-proof is again denoted by (†). A trace from the companion to the bud is denoted by the underlined formulas, with a progress point at the displayed application of (Case `ls`). As in the previous example, there is only one infinite path, and one can easily observe that the required infinitely progressing trace is obtained by concatenating copies of the displayed trace. So this pre-proof is indeed a proof.

We remark that the standard $\mathrm{LBI_{ID}}$ proof of $\mathtt{ls}\,x\,x', \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y$ proceeds by induction on $\mathtt{ls}\,x\,x'$ using the induction variables $z, z'$ (say) and the induction hypothesis $\mathtt{ls}\,z'\,y \mathrel{-\!\!*} \mathtt{ls}\,z\,y$, thus requiring a generalisation similar to that needed in Example 3.8.

**Proposition 4.11.** *It is decidable whether a $\mathit{CLBI}^{\omega}_{\mathit{ID}}$ pre-proof is a proof.*

*Proof.* (Sketch) The property of every infinite path posessing an infinitely progressing trace along a tail is an $\omega$-regular property, and hence reducible to the emptiness of a Büchi automaton. A full proof (for a general notion of trace) appears in [9]; a similar argument appears in [24]. □

## 5 Conclusions and Future Work

In this paper, we extend BI with a fairly general class of inductive definitions, and develop sequent calculus proof systems for formal reasoning in the resultant extension $\mathrm{BI_{ID}}$, as is needed in order to develop proper theorem proving support for inductive reasoning in separation logic. We hope that the formal framework(s) we present here will be of use to researchers in static analysis in providing a sound foundation for logical reasoning in future program verification applications employing inductively defined predicates (in a BI / separation logic context). In a technical sense, our contribution is a reasonably straightforward extension of the framework for inductive definitions and corresponding proof systems given in our previous work for first-order logic with inductively defined relations [8,9,10]. Thus one might reasonably hope that the key proof-theoretic results from that work, including appropriate completeness and cut-elimination theorems, will also extend to the systems we consider here. Certainly we expect that our cyclic proof system $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ subsumes the induction system $\mathrm{LBI_{ID}}$ (although we have not checked this in detail), with the question of their equivalence presenting similar difficulties to those discussed in [8,9,10]; in the setting of first-order logic with inductively defined relations, we have conjectured but not yet proven the equivalence of the two proof styles.

It is worth remarking that our logic $\mathrm{BI_{ID}}$ and the corresponding proof systems should be straightforwardly extensible to a more powerful definitional framework than the one we give here, for example by allowing inductive predicates to occur "negatively" in inductive definitions, subject to an appropriate stratification of predicates to ensure monotonicity as in iterated inductive definitions (cf. [17]).

One particularly promising avenue for further development is the development of static analysis applications based upon cyclic proof in separation logic. For example, our current work with Calcagno and Bornat develops a calculus for giving cyclic proofs of program termination in a (very) simple programming language, based upon a proof system of Hoare judgements which express termination from a given program point and under a given precondition [11]. We hope that it will also be possible to directly formulate cyclic proof systems for the verification of other properties. For example, such systems might use appropriate cyclic proof principles to establish invariants for the looping constructs in programs, or, given such invariants, to prove appropriate postconditions. An important factor related to such developments is the potential of cyclic proof for automated proof search. We have seen simple examples in which cyclic proof avoids the generalisation apparently necessary in the corresponding inductive proof (Examples 4.9 and 4.10); more generally, cyclic proof should offer a "least-commitment" approach to proof search, whereby the induction schema, variables and hypotheses are not chosen at the beginning of the proof, as in traditional inductive theorem proving, but are eventually selected implicitly via the satisfaction of the soundness condition. It would be interesting to examine the implications of these phenomena for proof search; we have previously given proof-theoretic machinery for analysing and manipulating the structure of general cyclic proofs [8,9] which may be of assistance in such investigations.

## References

1. Aczel, P.: An introduction to inductive definitions. In: Barwise, J. (ed.) Handbook of Mathematical Logic, North-Holland, pp. 739–782 (1977)
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
4. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.W.: Automatic termination proofs for programs with shape-shifting heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 386–400. Springer, Heidelberg (2006)
5. Biering, B., Birkedal, L., Torp-Smith, N.: BI hyperdoctrines and separation logic. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 233–247. Springer, Heidelberg (2005)
6. Birkedal, L., Torp-Smith, N., Reynolds, J.C.: Local reasoning about a copying garbage collector. In: Proceedings of POPL'04, pp. 220–231 (2004)
7. Bornat, R., Calcagno, C., O'Hearn, P.: Local reasoning, separation and aliasing. In: Proceedings of SPACE'04 (January 2004)
8. Brotherston, J.: Cyclic proofs for first-order logic with inductive definitions. In: Beckert, B. (ed.) TABLEAUX 2005. LNCS (LNAI), vol. 3702, pp. 78–92. Springer, Heidelberg (2005)
9. Brotherston, J.: Sequent Calculus Proof Systems for Inductive Definitions. PhD thesis, University of Edinburgh (November 2006)
10. Brotherston, J., Simpson, A.: Complete sequent calculi for induction and infinite descent. In: Proceedings of LICS-22 (to appear)

11. Brotherston, J., Calcagno, C., Bornat, R.: Cyclic proofs of termination in separation logic. Forthcoming
12. Bundy, A.: The automation of proof by mathematical induction. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning. vol. I, ch. 13, pp. 845–911. Elsevier Science, North-Holland, Amsterdam (2001)
13. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 182–203. Springer, Heidelberg (2006)
14. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
15. Guo, B., Vachharajani, N., August, D.I.: Shape analysis with inductive recursion synthesis. In: Proceedings of PLDI'07 (to appear)
16. Ishtiaq, S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: Proceedings of POPL'01 (January 2001)
17. Martin-Löf, P.: Haupstatz for the intuitionistic theory of iterated inductive definitions. In: Fenstad, J.E.(ed.) Proceedings of the Second Scandinavian Logic Symposium. North-Holland, pp. 179–216 (1971)
18. Nguyen, H.H., David, C., Qin, S., Chin, W.-N.: Automated verification of shape and size properties via separation logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, Springer, Heidelberg (2007)
19. O'Hearn, P.W., Pym, D.J.: The logic of bunched implications. Bulletin of Symbolic Logic 5(2), 215–244 (1999)
20. Pym, D.: The Semantics and Proof Theory of the Logic of Bunched Implications. In: Applied Logic Series, Kluwer, Dordrecht (2002)
21. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of LICS-17 (2002)
22. Élodie-Jane, S.: Extending separation logic with fixpoints and postponed substitution. Theoretical Computer Science 351(2), 258–275 (2006)
23. Sprenger, C., Dam, M.: On the structure of inductive reasoning: circular and tree-shaped proofs in the $\mu$-calculus. In: Gordon, A.D. (ed.) ETAPS 2003 and FOSSACS 2003. LNCS, vol. 2620, pp. 425–440. Springer, Heidelberg (2003)
24. Sprenger, C., Dam, M.: A note on global induction mechanisms in a $\mu$-calculus with explicit approximations. Theoretical Informatics and Applications (July 2003)
25. Yang, H.: An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In: Proceedings of SPACE 2001 (2001)

# Optimal Abstraction on Real-Valued Programs

David Monniaux

Laboratoire d'informatique de l'École normale supérieure
45, rue d'Ulm, 75230 Paris cedex 5, France

**Abstract.** In this paper, we show that it is possible to abstract program fragments using real variables using formulas in the theory of real closed fields. This abstraction is compositional and modular. We first propose an exact abstraction for programs without loops. Given an abstract domain (in a wide class including intervals and octagons), we then show how to obtain an optimal abstraction of program fragments with respect to that domain. This abstraction allows computing optimal fixed points inside that abstract domain, without the need for a widening operator.

## 1 Introduction

In program analysis, it is often necessary to derive relationships between program variables automatically; thus, within the framework of abstract interpretation [5], many *numerical abstract domains* have been developed. In addition to finding relationships between the values of program variables at a certain point, one may also want relationships between the input and the output variables of a program fragment, for instance procedures, so as to obtain a compositional and modular analysis. This paper presents algorithms for obtaining formulas expressing such relationships, at various degrees of precisions, as well as algorithms for extracting numerical results from these formulas. We consider programs operating on real variables — variables whose values lie in the real field ($\mathbb{R}$).

Relational abstract domains — those considering relations between several variables, as opposed to information on each variable separately — are often designed assuming integer ($\mathbb{Z}$), rational ($\mathbb{Q}$) or real ($\mathbb{R}$) variables inside the program to be studied. Why consider programs with real variables, which are not implementable in practice, while real-life program variables are generally either integer, or floating-point? This is because these relational domains suppose strong algebraic structures (ordered rings and fields) that floating-point numbers do not have (addition is not even associative). Then, real variables can be used as a sound *model* or *abstraction* of floating-point variables, enabling the analysis of programs using floating-point variables.

While it is unsound to assume that floating-point computations behave identically to real number computations, one can model floating-point computations using non-deterministic real number computations: each floating-point computation computes a result close to the ideal result, and the rounding error can be bounded. For instance, assuming IEEE-754 floating-point, the behavior of floating-point addition $x \oplus y$ is over-approximated by the real computation

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 104–120, 2007.

$x \oplus y = x + y + \epsilon$ with $|\epsilon| \leq \epsilon_r |x+y| + \epsilon_a$ chosen non-deterministically, where $\epsilon_r$ and $\epsilon_a$ are nonnegative coefficients depending on the floating-point type used. [9] Thus, relational abstract domains suitable for programs over real numbers are a worthy tool for the study of programs with floating-point computations.

This paper proposes an interpretation of real-valued programs without loops as formulas in the theory of real closed fields. [3][1, Chapter 2] This interpretation captures perfectly the input-output relationships of such programs. Furthermore, if one is given preconditions of the form $f_j(v_1, \dots, v_n) \leq c_j$ where $v_k$ are the input values of the variables of the programs, the $f_j$ are polynomials and $c_j$ some coefficient, then one can obtain, automatically and with arbitrary precision, $c'_j$ such that $f_j(v'_1, \dots, v'_n) \leq c'_j$ where $v'_k$ are the output values of the variables of the program, and these $c'_j$ are optimal bounds. Such pre-conditions and post-conditions encompass a wide variety of abstract domains, including intervals, octagons, octahedra etc. [8,4], meaning that whole programs without loops can be algorithmically optimally approximated with respect to these abstract domains.

In the case of programs with loops, we cannot hope to have such a result, which would entail solving the halting problem. However, we show how to obtain a formula defining the exact least fixed point of the optimal abstraction of the program semantics. Again, we can then obtain numerical values with arbitrary precision. No “widening operator” is used, and the loss of precision entirely depends on the choice of constraints representable by the abstract domain.

In section 2, we shall recall the definition and classical results of the theory of real closed fields, and then we shall propose algorithms for bounding, with arbitrary precision, real numbers defined by formulas in that theory. In section 3, we shall define the language that we consider and give an “exact” abstract semantics of this language using formulas in the theory of real closed fields. In section 4, we shall see how to get the optimal abstractions we announced.

# 2 Mathematical Preliminaries

Throughout the paper, we shall express relationships between real numbers as formulas within the theory of real closed fields (polynomial equalities and inequalities, logical connectors, quantifiers). This theory is powerful yet decidable. We shall also obtain some real numbers as the unique solution (or model) of a formula in that theory; we shall show that from such a characterization we can compute approximations (lower and upper bounds) with arbitrary precision.

## 2.1 Real Closed Fields and Algebraic Numbers

Let us recall the definition of the syntax and semantics of the theory of real closed fields. In the rest of the paper, by “formula” we shall mean a formula in that theory.

We consider the following formulaic language: atomic formulas are of the form $P(x, y, z, \dots) \bowtie C$ where $C$ is a rational number, $P$ is a polynomial with rational coefficients, and $\bowtie$ is a comparison operator ($<$, $\leq$, $=$, $\neq$, $\geq$, $>$); compound formulas are formed using logical connectors $\neg$, $\wedge$, $\vee$, as well as quantifiers $\exists x$ and $\forall x$ where $x$ is a variable. If a formula contains no quantifier, it is *quantifier-free*. The set of formulas on the set of variables $V$ is noted $\mathcal{F}(V)$, while the set of quantifier-free formulas is noted $\mathcal{F}_{\mathrm{QF}}(V)$.

Notions of free and bound variables are defined as usual. Model candidates $m$ of a formula $F$ are *assignments* for the free variables of $F$, an assignment being a map from the set $FV(F)$ of free variables of $F$ to the reals. We say that a model candidate $m$ is a *model* of $F$, and note $m \models F$, with the obvious definition. The set of models of $F$ is noted $\mathcal{M}(F)$. We recall the following result: [1, Th. 2.77].

**Theorem 1 (Tarski, 1951).** *There exists an algorithm $E$ such that for any formula $F$ in the above language, the algorithm outputs a quantifier-free formula $E(F)$ such that $FV(E(F)) = FV(F)$ and, for any model candidate $m$, $m \models F$ if and only if $m \models E(F)$.*

In practice, one does not use Tarski's algorithm, which has nonelementary complexity, but one rather uses cylindrical algebraic decomposition, which has "only" $2^{2^n}$ complexity in the size of the formula. [3][1, Ch. 11].

In many cases, we shall have a set $V$ of "pre" variables and a set $V'$, a disjoint copy of $V$, of "post" variables; a variable $x$ in $V$ corresponds to a copy $x'$ in $V'$; we call formulas over such variables *pre-post formulas*. These will encode input-output relationships of program fragments. To an assignment $a \in \mathbb{R}^V$ we associate an assignment $a' \in \mathbb{R}^{V'}$. Given a pre-post formula $F$ and a set of assignments $A \subseteq \mathbb{R}^V$, we define associated "predicate transformers":

$$\overrightarrow{F}(A) = \{b \mid \exists a \in A \ (a, b') \models F\} \tag{1}$$

$$\overleftarrow{F}(A) = \{b \mid \exists a \in A \ (b, a') \models F\} \tag{2}$$

We shall compute using algebraic numbers. An algebraic number $r$ will be specified as a formula $\tilde{r}$ in the theory of real closed fields, with a single free variable $x$, such that $x \mapsto r$ is the model of $\tilde{r}$. We can restrict the formulas to conjunctions of polynomial equalities and inequalities without loss of generality: if $F$ is a quantifier-free formula with a single assignment $x \mapsto r$ as model, it can algorithmically be put in disjunctive normal form $C_1 \vee \cdots \vee C_n$ where $C_1, \ldots, C_n$ are conjunctions; then inconsistent conjunctions $C_i$ can be removed algorithmically; any of the remaining conjunctions will have a single model $x \mapsto r$ and fits our needs. Algebraic numbers specified in this way can be algorithmically approximated to arbitrary precision, within a framework of *computable reals*, as shown in the following section.

### 2.2 Computable Reals

Our abstract domains will "compute" reals in an indirect way: instead of computing the value of a real number (which is impossible to do exactly in most cases), the abstract domain will define it as the unique solution of a quantifier-free formula with one variable; for instance, $\sqrt{2}$ would be defined as the unique $x$ such that $x^2 = 2 \wedge x > 0$. In this section, we show that given such a characterization, one can algorithmically bound the real number with arbitrary precision; that is, given $\epsilon \in \mathbb{Q}$, $\epsilon > 0$, obtain $m, M \in \mathbb{Q}$ such that $m \leq x \leq M$ and $M - m \leq \epsilon$. More generally, we shall show that for any quantifier-free formula in the theory of real closed fields with one free variable, we can obtain a finite description of its domain of validity, such that all numbers used inside the description are computable with arbitrary precision.[1]

We define computable reals through *approximation functions*: instead of a real $r$, which cannot be represented directly in a machine, we shall consider a computable function $\tilde{r}$ taking a positive rational number $\epsilon$ as a parameter and outputting a couple $(m, M)$ of rational numbers such that $M - m \leq \epsilon$ and $m \leq r \leq M$, called an *$\epsilon$-approximation*.[2]

We shall give our algorithms in a "literate programming" or "proof-carrying code" fashion, mixing each algorithm with a proof of its correctness. For the sake of simplicity, we preferred to give all algorithms "from scratch" instead of relying on advanced techniques.[3]

Let $a$ and $b$ be computable reals given by approximation functions $\tilde{a}$ and $\tilde{b}$. It is straightforward to compare these two numbers, provided that we know that they are different.

**Algorithm** Compare: Compare two computable reals known to be different

If $a \neq b$, then there is an algorithm that decides whether $a < b$ or $a > b$ given approximation functions $\tilde{a}$ and $\tilde{b}$: start with $\epsilon = 1$; compare the intervals $\tilde{a}(\epsilon)$ and $\tilde{b}(\epsilon)$; if they do not overlap, the case is settled, otherwise divide $\epsilon$ by 10 and try again. The algorithm will terminate at the latest when $\epsilon < |b - a|/2$.

This algorithm loops forever if $a = b$. Throughout the rest of this section, we shall take precautions so that we never use Compare on operands that could be equal.[4] We then define elementary arithmetic operators over approximation functions:

**Algorithm** Plus: Add two computable reals

$a+b$ is also a computable real: for $\epsilon > 0$, compute $\epsilon/2$-approximations $[m_a, M_a]$ of $a$, $[m_b, M_b]$ of $b$, and output $[m_a + m_b, M_a + M_b]$ as a $\epsilon$-approximation of $a + b$.

[1] This is a generalization of a result of Turing, that real algebraic numbers are computable [14, §1.vi].

[2] Turing's original characterization of the class of computable reals [14] [15, Def. 4.1.12] used machines that enumerated the decimals of the number. The class of computable reals defined in this fashion is identical to ours, but there are drawbacks to this representation: it may be necessary in order to compute the $n$-th digit of a result to go arbitrarily far in the representation of the operands. We thus rather use a representation very close to that of Weihrauch. [15, §1.3.2].

[3] Alternatively, the same result may be reached using published algorithms [1, Alg. 10.4 to 10.17] for isolating roots of polynomials, pairs of polynomials or finding the sign of a polynomial at the roots of another, together with a dichotomy solving method.

[4] This is actually an essential restriction of any representation of computable reals. [15, Th. 4.1.16].

A similar algorithm works for $a - b$, defining MINUS.

**Algorithm** MULT: Multiply two computable reals

- compute a 1/2-approximation $[m_a, M_a]$ of $a$; if $0 \in [m_a, M_a]$, then write $a \cdot b = (a+1) \cdot b - b$ and the problem is reduced to the case where $a$ cannot be zero;
- decide whether $a < 0$ or $a > 0$ by testing whether $m_a < 0$: if $a < 0$, write $a \cdot b = -((-a) \cdot b)$ and the problem is reduced to the case where $a > 0$; we also have computed a upper bound $L_a \in \mathbb{Q}$ of $a$;
- do similarly with $b$ and the problem is reduced to the case where $b > 0$; we also have computed a upper bound $L_b \in \mathbb{Q}$ of $b$;
- compute a $\epsilon/(2L_b)$-approximation $[m_a, M'_a]$ of $a$ and a $\epsilon/(2L_a)$-approximation $[m_b, M'_b]$ of $b$; let $M_A = \min(M'_A, L_a)$ and $M_b = \min(M'_b, L_b)$ and output $[m_a m_b, M_a M_b]$; this is a $\epsilon$-approximation of $a.b$ since $M_a M_b - m_a m_b = M_a(M_b - m_b) + m_b(M_a - m_a) \leq \epsilon$.

Now for three algorithms that will be later used as subroutines:

**Algorithm** DECIDESIGN: Decide the sign of $P(x)$ if $P(x) \neq 0$

It follows that if $P \in \mathbb{Q}[X]$, and $r$ is a real given by $\tilde{r}$ such that $P(r) \neq 0$, then we can decide whether $P(r) < 0$ or $P(r) > 0$ using PLUS and MULT over the polynomial structure, then COMPARE.

**Algorithm** FINDROOT: Find the unique root of $P$ in an interval $[r_1, r_2]$ of monotonicity

Let $r_1 < r_2$, given by $\tilde{r}_1$ and $\tilde{r}_2$, and $P$ a polynomial such that $P$ is strictly increasing over $[r_1, r_2]$, $P(r_1) < 0$ and $P(r_2) > 0$. Let $\epsilon > 0$. Compute $[m_1, M_1]$ a $\epsilon$-approximation of $r_1$ and $[m_2, M_2]$ an $\epsilon$-approximation of $r_2$. If $P(M_1) \geq 0$, then $[m_1, M_1]$ is an $\epsilon$-approximation of $r_0$. If $P(m_2) \leq 0$, then $[m_2, M_2]$ is a $\epsilon$-approximation of $r_0$. We thus suppose $P(M_1) < 0$ and $P(m_2) > 0$ and apply a dichotomy algorithm between the two, until we reach the desired precision.

**Algorithm** FINDROOTINF: Find the unique root of $P$ in an interval $(-\infty, r_2]$ of monotonicity[5]

If we know that $P$ is strictly increasing on $(-\infty, r_2]$, $P(r_2) > 0$, noting $r$ the root of $P$ such that $r < r_2$, then, similarly, let $\epsilon > 0$; compute $[m_2, M_2]$ a $\epsilon$-approximation of $r_2$; if $P(m_2) \leq 0$ then $[m_2, M_2]$ is a $\epsilon$-approximation of $r$. If $P(m_2) > 0$ then take $k \in \mathbb{N}$, $-k < m_2$, $k$ increasing until $P(-k) < 0$; then apply the dichotomy algorithm between $-k$ and $m_2$.

Let us recall a familiar result, which we shall use with $K = \mathbb{Q}$ and $K' = \mathbb{R}$:

**Lemma 1.** *Let $K$ be a field and $K'$ an extension of $K$. If $\xi \in K'$ is a common root of nonzero polynomials $P$ and $Q$ from $K[X]$, then it is a root of their greatest common divisor* $\gcd(P, Q)$ *in $K[X]$. Thus, co-prime polynomials have no common root.*

*Proof.* $K[X]$ is a principal ring [7, Ch. 4, Th. 1.2], there exist polynomials $A$ and $B$ in $K[X]$ such that $\gcd(P, Q) = A.P + B.Q$. The result follows by applying both members of the equation to $\xi$.

[5] We note open intervals $(a, b)$, closed intervals $[a, b]$.

Several of our algorithms operate on *sign diagrams*. A sign diagram for a nonzero polynomial $P \in \mathbb{Q}[X]$ is a sequence $-, \tilde{r}_1, +, \tilde{r}_2, -, \tilde{r}_3, +, \ldots, \tilde{r}_n, +$, where the $\tilde{r}_i$ are approximation functions for the roots of $P$. Such a diagram means that the polynomial function $P(x)$ is negative for large negative $x$, then passes a root $r_1$ that can be approximated to arbitrary precision by $\tilde{r}_1$, then becomes positive, etc.

Sign diagrams for polynomials of degrees 0 and 1 are straightforward to compute, as are the first and final signs of the diagram for any polynomial, which are obtained from the parity of the degree of the polynomial and the sign of the leading coefficient. Given the diagram of $P$ and a nonnegative exponent $e$, it is straightforward to compute the diagram for $P^e$; and given the diagram for $P$ and a coefficient $a \in \mathbb{Q}$, it is also straightforward to compute the diagram for $aP$.

Given two polynomials $P$ and $Q$ with no common roots, one obtains the sign diagram for $P.Q$ through a simple sorted list merging procedure using Compare. This algorithm, however, does not apply in case $P$ and $Q$ have common roots. We use the fact (Lem. 1) that the common roots of $P$ and $Q$ are the roots of the greatest common divisor $\gcd(x, y)$ of these polynomials to work around this difficulty. $\gcd(x, y)$ can be computed using Euclid's algorithm.

**Algorithm** SignDiagram: Compute the sign diagram of a polynomial

We shall now show how to compute the sign diagram of a polynomial $P$ by induction on the degree $n$ of $P$. We have already noted that it is trivial to compute diagrams for polynomials of degrees 0 and 1. We now shall suppose that we can compute the sign diagrams of polynomials of degree less than $n$, and show that we can compute the sign diagram of a polynomial of degree $n$. First, define a subroutine:

**Algorithm** SignDiagramProduct:

Take as input a list $(P_1, e_1), \ldots, (P_m, e_m)$ of couples each formed of a polynomial of degree less than $n$ and a positive exponent, output the sign diagram of the product $P_1^{e_1} \times \cdots \times P_m^{e_m}$. We proceed by induction on the sum of the degrees of $P_1, \ldots, P_m$. If this sum is 0 or 1, then the case is trivial.

- Check whether there exist $P_i$ and $P_j$ $(i \neq j)$ not co-prime; if so, compute $Q_i = P_i / \gcd(P_i, P_j)$ and $Q_j = P_j / \gcd(P_i, P_j)$, then replace $(P_i, e_i)$ and $(P_j, e_j)$ by $(Q_i, e_i)$, $(Q_j, e_j)$, $(\gcd(P_i, P_j), e_i + e_j)$ in the list. The sum of the degrees has decreased by the degree of $\gcd(P_i, P_j)$, but the product $P_1^{e_1} \times \cdots \times P_m^{e_m}$ has stayed the same, and thus we can solve the problem through a recursive call.
- Otherwise, the $P_i$ are pairwise co-prime. Since they all have degree less than $n$, we can obtain their sign diagrams. We then apply the exponent algorithm, then the algorithm for the sign diagrams of a product of polynomials with no common roots.

Consider now a polynomial $P$ of degree $n$.

- If $P$ and its derivative $P'$ are not co-prime, then let $Q = P / \gcd(P, P')$. $Q$ and $\gcd(P, P')$ will have degree at most $n - 1$, so we can invoke SignDiagramProduct and obtain the sign diagram of their product $P$.

- If they are co-prime: $P$ only has single roots. Compute the sign diagram of $P'$, which gives us intervals of monotonicity for $P$. Then, compute the sign diagram of $P$ as follows:
    - The leftmost sign is deduced from the leading coefficient and parity of the degree of $P$. Without loss of generality, we shall suppose it is positive.
    - Compute the sign of $P(r_1)$ (using DecideSign) where $r_1$ is the first root in the sign diagram of $P'$; this is possible because $r_1$ is not a root of $P$. If it is negative, search for a root of $P$ to the left of $r_1$ using FindRootInf.
    - For each subsequent root $r_k$ of $P'$, compute the sign of $P(r_k)$ (using DecideSign), and if it is different from the sign of $P(r_{k-1})$, search for a root of $P$ in $[r_{k-1}, r_k]$ using FindRoot.

For a system $S$ of polynomial equalities or inequalities over a real variable $x$, we call *validity diagram* a sequence $b_0, r_1(B_1), b_1, r_2(B_2), \ldots, r_m(B_m)$ where $r_1, \ldots, r_m$ are given by approximation functions $\tilde{r}_1, \ldots, \tilde{r}_m$, and the $b_i$ and $B_i$ are booleans; $b_0$ says whether $S$ is always or never satisfied over $(-\infty, r_1)$, $B_1$ whether $S$ is satisfied at $r_1$, $b_1$ whether $S$ is always or never satisfied over $(r_1, r_2)$ and so on.

**Algorithm** Domain: Domain of validity of a quantifier-free formula with one free variable

Consider now a quantifier-free formula $F$ with one free variable, made up of of polynomial equalities and inequalities $P_i \bowtie 0$. Similarly as in SignDiagramProduct, take greatest common divisors until obtaining a base $B_k$ of pairwise co-prime polynomials such that for all $i$, $P_i$ can be written $P_i = B_1^{e_1} \times \cdots \times B_m^{e_m}$. Compute the sign diagrams of all $B_k$. The validity diagram of $F$ can be computed from the $B_k$ using, as previously, a variant of the merging of sorted lists and the fact the $B_k$, pairwise, have no common roots.

By preprocessing formulas through quantifier elimination, we can algorithmically approximate to arbitrary precision any (algebraic) real defined by a formula in the theory of real closed fields.

**Corollary 1.** *If $F$ is a formula of the theory of real closed fields with one free variable, such that $F$ defines a single real, then this real is algebraic and can be algorithmically approximated to arbitrary precision.*

## 3 Concrete and Exact Abstract Semantics

We consider a simple block-structured programming language without loops, and a concrete semantics as the binary relation between input variables and output variables. This concrete semantics can be exactly represented using formulas in the theory of real closed fields.

### 3.1 Concrete Semantics

We consider the following language $L$:

- Real expressions are constructed over: real variables (taken in a set $V$ of variable names), arithmetic operators ($+$, $-$, $/$, $\times$), integer constants.

- Boolean expressions are constructed from atomic formulas using $\vee$, $\wedge$, $\neg$
- Atomic formulas are constructed from real expressions and relational operators ($<$, $>$, $=$, $\leq$, $\leq$, $\geq$)
- The only control construct is if-then-else, where the condition is a boolean expression.
- The only instruction is the assignment $x := e$, where $x$ is a real variable and $e$ a real expression.
- There is a nondeterministic choice instruction $x := [m, M]$, choosing $x$ between $m$ and $M$.

A program in $L$ has a concrete semantics as a binary relation over $\mathbb{R}^V$: $(\mathit{pre}, \mathit{post}) \in [\![P]\!]$ if it is possible to reach the state *post* at the end of the execution of $P$ starting from the state *pre*.

For the sake of simplicity, we shall consider the sub-language $L_s$ where expressions in assignments are to be simple, with only a single operator, and arithmetic expressions in boolean expressions are to be variables; programs in $L$ can be turned into equivalent programs in $L_s$ using additional variables to store intermediate results.

It is immediate that all results in the rest of the paper persist if one replace the theory of real close fields with another arithmetic theory admitting quantifier elimination (Presburger arithmetic, or the theory of rational or real inequalities).

### 3.2 Quantifier-Free Closed Real Field Formulas

We compile programs without loops, compositionally, into quantifier-free formulas from the theory of real closed fields such that for a program $P$, the formula $[\![P]\!]_F$ models the input-output relationship $[\![P]\!]$ exactly.

We consider disjoint copies $V'$ and $V_\exists$ of the set $V$: $V$ will be used for free formula variables denoting the input values of program variables, $V'$ for free variables denoting the output values of program variables, and $V_\exists$ for variables bound by existential quantifiers, which are to be removed from the formulas by quantifier elimination. $[\![]\!]_F$ is defined as follows:

**Arithmetics Addition** $[\![a := b + c]\!]_F \triangleq a' = b + c$

**Subtraction** $[\![a := b - c]\!]_F \triangleq a' = b - c$

**Multiplication** $[\![a := b * c]\!]_F \triangleq a' = b \times c$

**Division** $[\![a := b/c]\!]_F \triangleq b = a' \times c$

**Tests** $[\![\texttt{if } c \texttt{ then } p_1 \texttt{ else } p_2]\!] \triangleq (c \wedge [\![p_1]\!]_F) \vee (\neg c \wedge [\![p_2]\!]_F)$

**Composition** $[\![P_1; P_2]\!]_F \triangleq E(\exists v_1 \ldots \exists v_n \ f_1 \wedge f_2)$ where $f_1$ is $[\![P_1]\!]_F$ where all variables in $V'$ have been replaced by their copy in $V_\exists$, $f_2$ is $[\![P_2]\!]_F$ where all variables in $V$ have been replaced by their copy in $V_\exists$, and $v_1, \ldots, v_n$ are the free variables of $f_1$ and $f_2$ that are in $V_\exists$. This is the only place where we need quantifier elimination.

**Nondeterministic choice** $[\![x := [m, M]]\!]_F \triangleq m \leq x' \leq M$

**Proposition 1.** *Let $P$ be a program in $L_s$. Let pre, post $\in \mathbb{R}^V$. Then (pre, post) $\in [\![P]\!]$ if and only if (pre, post$'$) $\models [\![P]\!]_F$.*

Alternatively, we could allow formulas including quantifiers and defer quantifier elimination until it is actually needed.

What happens with programs with loops? On programs operating over integers, one can obtain logical relations linking inputs and outputs: sequences of values of program variables across iterations can be encoded into couples of integers using Gödel's $\beta$ function [16, Chapter 7], and one can thus construct a finite formula defining the strongest loop invariant. [6] Then, of course, there is no way to decide the formulas obtained.

In the case of programs with reals, the situation is different. There exist some programs such that there is no formula in the theory of real closed fields that precisely links the inputs and the outputs. Consider, for instance:

```
s=0; x=1; k=1;
while (k < n) { x=x/k; s=s+x; k=k+1; }
```

As $n \to \infty$, the output $s$ of this program increases and tends to exp(1).

Let us suppose that there exists a relationship $P(n, s')$ in the theory of real closed fields between the initial value of $n$ and the final value of $s$. The formula

$$(\forall n \exists s' \; P(n, s') \Rightarrow s' \leq l) \wedge (\forall b \; (\forall n \exists s' \; P(n, s') \Rightarrow s' \leq b) \implies l \leq b). \qquad (3)$$

defines a single real, the least upper bound of the possible outputs, which is exp(1). By Cor. 1, this real is algebraic; but it is well-known that exp(1) is transcendental, a contradiction. Thus, in general, strongest loop invariants cannot be expressed within the theory of real closed fields. We shall thus aim at expressing some kind of loop invariant, not necessarily the strongest.

## 4 Optimal Abstraction over Polynomial Constraint Domains

We now consider the abstraction of program states (in $\mathbb{R}^V$) using domains defined by polynomial constraints. This family of domains includes many "classical" numerical abstract domains. We shall show that, with respect to such domains, optimal abstractions are computable (in the sense of: one can compute approximations to arbitrary precision) for programs without loops. Furthermore, we shall show that least fixed points in such domains are also computable; we thus obtain a semantics for loops, optimal in a certain sense. Finally, we shall see how to define a general computable abstract semantics for programs with loops.

[6] A similar construction proves Cook's theorem: Floyd-Hoare axiomatic semantics on programs using integers is complete with respect to Peano's arithmetic [16, Th. 7.5].

Throughout this section, we shall be concerned with the forward propagation problem: given an abstract precondition $s^\sharp$ and a program fragment $P$, characterize an abstract postcondition $e^\sharp$ such that $\overrightarrow{\llbracket P \rrbracket} \circ \gamma_f(s^\sharp) \subseteq \gamma_f(e^\sharp)$ ("if this precondition holds, then this postcondition must also hold"), and in particular an optimal $e^\sharp$ in a certain sense. However, all results work if one seeks to compute preconditions, using $\overleftarrow{\llbracket P \rrbracket}$ ("if this postcondition holds, then the input of the program must have fit this precondition").

### 4.1 Polynomial Constraint Domains: Definition

A polynomial constraint domain $D_f^\sharp$ is defined by a family $(f_\lambda)_{\lambda \in \Lambda}$ of polynomials,[7]with variables in $V$. An element of $D_f^\sharp$ is either $\perp$, either a vector in $(-\infty; +\infty]^\Lambda$ of *parameters*. (We assume $\Lambda \neq \emptyset$).[8]

The $\gamma_f : D_f^\sharp \rightarrow \mathcal{P}\left(\mathbb{R}^V\right)$ maps each element of $D_f^\sharp$ to the set of program states that it represents, that is, its *concretization*: $\gamma_f(\perp) = \emptyset$, and $\gamma_f((x_\lambda)_{\lambda \in \Lambda})$ is the set of variable assignments $a \in \mathbb{R}^V$ such that for all $\lambda$, $f_\lambda(a) \leq x_\lambda$. We exclude from $D_f^\sharp$ vectors $x$ such that $\gamma(x)$ would be empty, that is, vectors specifying inconsistent constraints; we do so in order to have $\perp$ as the sole representation of the empty set.

Several "classical" numerical domains can be interpreted as polynomial constraint domains:

**Intervals:** Polynomials are $v$ and $-v$, for all $v \in V$.
**Difference matrices:** Polynomials are $v_1 - v_2$, for all $v_1, v_2 \in V$.
**Octagons:** Polynomials are $\pm v$, for all $v \in V$, and $\pm v_1 \pm v_2$, for all $v_1, v_2 \in V$. [8]
**Octahedra:** Polynomials are $\pm v_1 \pm v_2 \pm \cdots \pm v_n$, for all $v_1, v_2, \ldots, v_n \in V$. [4]
**Template linear constraints:** Restriction to linear polynomials. [13]

Such a domain can be fitted with a straightforward complete lattice structure $(\sqsubseteq, \sqcup, \sqcap)$, making $\gamma_f$ increasing with respect to $\sqsubseteq$ and $\subseteq$:

- $\perp$ is the unique least element;
- $x \sqsubseteq y$, $x, y \in (-\infty, +\infty]^\Lambda$, if for all $\lambda \in \Lambda$, $x_\lambda \leq y_\lambda$;
- the least upper bound and greatest lower bounds of a family of vectors are defined coordinate-wise.

We can also provide an optimal abstraction function[9] $\alpha_f$ such that $(\alpha_f, \gamma_f)$ form a Galois connection [5, §4.2.2]: $\alpha_f(\emptyset) = \perp$; $\alpha_f(S)$ (where $S \neq \emptyset$) is the

[7] Polynomials are used to define constraints of the form $f_\lambda(v_1, \ldots, v_n) \leq d_\lambda$. More generally, our framework applies to any predicate $P_\lambda(v_1, \ldots, v_n; d_\lambda)$ of the theory of real closed fields, such that the set of models $\mathcal{M}(d_\lambda)$ for the $v_1, \ldots, v_n$ is left-continuous with respect to the parameter $d_\lambda$: $\mathcal{M}(\inf D_\lambda) = \bigcap_{d_\lambda \in D_\lambda} \mathcal{M}(d_\lambda)$. All the results given in the following sections also apply to that extended framework.

[8] We can also consider an additional family of predicates without parameters, abstracted by their truth value.

[9] Neither $\alpha_f$ nor $\gamma_f$ are computable functions: they operate on unbounded sets of rational or real numbers. The purpose of this paper is to show how to compute certain quantities defined mathematically using $\alpha_f$ or $\gamma_f$.

vector $(x_\lambda)_{\lambda\in\Lambda}$ where $x_\lambda = \sup_{s\in S} f_\lambda(s)$. From its definition, it is obvious that $\alpha_f$ preserves least upper bounds. [5, Prop. 6]

$\alpha_f$ distinguishes among several possible abstractions of some set $X$ the abstraction $\alpha_f(X)$ such that $\alpha_f(X)$ is minimal. Not only does this avoid taking a non-optimal abstraction — if we chose $y^\sharp$ when there exists $x^\sharp$ such that $X \subseteq \gamma_f(x^\sharp) \subsetneq \gamma_f(y^\sharp)$, then $y^\sharp$ is a non-optimal choice as an abstraction — but it also provides for a "canonical" representation. Indeed, the concretization function $\gamma_f$ is injective in the case of the intervals, but needs not be so in general. In the case of the difference matrices and the octagons, there may be an infinity of abstract elements with the same concretization: if we have the constraints $v_1 - v_2 \leq C_{1,2}$, $v_2 - v_3 \leq C_{2,3}$, and $v_1 - v_3 \leq C_{1,3}$, then we get the same concretization as long as $C_{1,3} \leq C_{1,2} + C_{2,3}$. These domains, however, are fitted with a *reduction* or *closure* operation[10] such that they provide results that are minimal with respect to $\sqsubseteq$. In this example, the closure operation realizes that the constraint $v_1 - v_3 \leq C_{1,2} + C_{2,3}$ can be derived from the first two and can be used to refine the third one if $C_{1,2} + C_{2,3} < C_{1,3}$. The closure operation can also detect that some constraints are inconsistent and the result should be $\bot$.

Let $\psi : \mathcal{P}\left(\mathbb{R}^V\right) \to \mathcal{P}\left(\mathbb{R}^V\right)$. A function $\psi^\sharp : D_f^\sharp \to D_f^\sharp$ is said to be an *abstraction* of $\psi$ if $\forall d^\sharp \in D_f^\sharp,\ \psi \circ \gamma_f(d^\sharp) \subseteq \gamma_f \circ \psi^\sharp(d^\sharp)$. The *optimal abstraction* of $\psi$ is $\alpha_f \circ \psi \circ \gamma_f$. In program analysis in general, it is possible that this optimal abstraction is not computable. However, in the next sub-section, we shall show that this optimal abstraction is computable on programs without loops with the kind of domains that we consider here.

### 4.2 Optimal Abstraction Without Fixed Points

In section 3.2, we have shown that the input-output relationship of concrete program variables can be represented as a formula in the theory of real closed fields. Such a formula links the output value of the program variables to their input values. Here, we shall see that the optimal abstraction of a program fragment with respect to a polynomial constraint abstract domain can also be represented as a formula in that theory; that is, we shall give a formula linking the input and output parameters for these constraints.

Consider now a set of program states abstracted by $d^\sharp \in D_f^\sharp$, and a pre-post formula $\phi$ over $\mathbb{R}^V \times \mathbb{R}^{V'}$. We are interested in finding the optimal abstraction of $\overrightarrow{\phi}(d^\sharp)$. We will show that, in the case where $d^\sharp \neq \bot$, the coefficients of the vector defining this optimal abstraction $d^{\sharp\prime}$ are related to the coefficients of $d^\sharp$ through a formula of the theory of real closed fields, and that the cases where $\bot$ appear are settled by deciding a formula of that theory.

An element of $d^\sharp$ is either $\bot$, or a vector, indexed by $\lambda \in \Lambda$, of $d_\lambda \in \mathbb{R} \cup \{+\infty\}$. We use a representation using only real variables: $(d_b, (d_\lambda)_{\lambda\in\Lambda}, (\bar{d}_\lambda)_{\lambda\in\Lambda})$, where:

[10] The lower closure operation $\alpha_f \circ \gamma_f$ is defined for every Galois connection [5, §4.2.2], but it needs not be computable in general. In the case of difference matrices and octagons with rational coefficients, it is effectively computable by a shortest path algorithm, and certain operations require their operands to be closed. [8, §V.B].

- $d^\sharp = \bot$ is encoded by $d_b = 1$ and any other value elsewhere;
- $d^\sharp = (d^\sharp_\lambda)_{\lambda \in \Lambda}$ is encoded as follows: $d_b = 0$ and for all $\lambda \in \Lambda$, either $d^\sharp_\lambda < \infty$ and $d_\lambda = d^\sharp_\lambda$, $\bar{d}_\lambda = 0$, or $d^\sharp_\lambda = \infty$ and $\bar{d}_\lambda = 1$.

If $d^\sharp = \bot$, then this optimal abstraction is obviously $\bot$. For the sake of ease of notation, let $\Lambda = \{1, \dots, m\}$ and $V = \{v_1, \dots, v_n\}$. Let *abstracts*$(d, x)$ be the formula $\bar{d} = 1 \vee (\bar{d} = 0 \wedge x \leq d)$. Let *isNonEmpty*$(d^\sharp)$ be the formula $\exists v_1 \dots \exists v_n \;\; \mathit{abstracts}(d_1, f_1(v_1, \dots, v_n)) \wedge \dots \wedge \mathit{abstracts}(d_m, f_m(v_1, \dots, v_n))$; if *isNonEmpty*$(d^\sharp)$ is false then $\overrightarrow{\phi}(d^\sharp) = \emptyset$ and, again, $\bot$ is an optimal abstraction. *abstracts*$(d, x)$ may be decided algorithmically by quantifier elimination if the $d_1, \dots, d_m$ are rational numbers or algebraic numbers specified as in Sec. 2.2.

Let

$$\begin{aligned} \mathit{step}(\phi, d^\sharp) \triangleq d_b = 0 \wedge \exists v_1 \dots \exists v_n \;\; & \mathit{abstracts}(d_1, f_1(v_1, \dots, v_n)) \wedge \\ & \dots \wedge \mathit{abstracts}(d_m, f_m(v_1, \dots, v_n)) \wedge \phi. \end{aligned} \tag{4}$$

The models of $\mathit{step}(\phi, d^\sharp)$ are exactly the assignments for $v'_1, \dots, v'_n$ such that $(v'_1, \dots, v'_n) \in \overrightarrow{\phi} \circ \gamma_f(d^\sharp)$. Now, we need to define the image of that set by $\alpha_f$ using formulas.

The projections of those assignments over the $f_\lambda$ constraints are defined by:

$$\mathit{step}_\lambda(\phi, d^\sharp, x) \triangleq \exists v'_1 \dots \exists v'_n \;\; \mathit{step}(\phi, d^\sharp) \wedge x = f_\lambda(v'_1, \dots, v'_n) \tag{5}$$

The following formula has models $(d, \bar{d}) \models \mathit{isSup}(d, x, P)$ such that $\bar{d} = 1$ if $\{x \mid P(x)\}$ has no upper bound, and otherwise $\bar{d} = 0$ and $d = \sup\{x \mid P(x)\}$:

$$\begin{aligned} & \mathit{isSup}(d, x, P) \triangleq (\bar{d} = 1 \wedge \forall y \exists x \; y \leq x \wedge P(x)) \vee \\ & \quad (\bar{d} = 0 \wedge (\forall x \; P(x) \implies x \leq d) \wedge (\forall y (\forall x \; P(x) \implies x \leq y) \implies d \leq y) \end{aligned} \tag{6}$$

Thus, the formula for defining the optimal parameter for the constraint indexed by $\lambda$ is: $\mathit{supStep}_\lambda(\phi, d^\sharp, d') \triangleq \mathit{isSup}(d', x, \mathit{step}_\lambda(\phi, d^\sharp, x))$.

Finally, we define:

$$\begin{aligned} & \mathit{abstrStep}(\phi, d^\sharp, d'^\sharp) \triangleq (d'_b = 1 \wedge \neg \exists v'_1 \dots \exists v'_n \;\; \mathit{step}(\phi, d^\sharp)) \vee \\ & \qquad (d'_b = 0 \wedge \mathit{supStep}_1(\phi, d^\sharp, d'_1) \wedge \dots \wedge \mathit{supStep}_m(\phi, d^\sharp, d'_m)) \end{aligned} \tag{7}$$

We thus have lifted a formula $\phi$ between concrete states to an optimal formula $\mathit{abstrStep}(\phi, d^\sharp, d'^\sharp)$ between abstract states, and the following holds:

**Theorem 2.** *Let $\phi$ be an input-output formula over variables $V$. Then, the constructed formula $\mathit{abstrStep}(\phi, d^\sharp, d'^\sharp)$ has models $(d^\sharp, d'^\sharp) \models \mathit{abstrStep}(\phi, d^\sharp, d'^\sharp)$, where $d^\sharp = (d_b, (d_\lambda)_{\lambda \in \Lambda}, (\bar{d}_\lambda)_{\lambda \in \Lambda})$ and $d'^\sharp = (d'_b, (d'_\lambda)_{\lambda \in \Lambda}, (\bar{d}'_\lambda)_{\lambda \in \Lambda})$, exactly such that $d'^\sharp = \alpha_f \circ \overrightarrow{\phi} \circ \gamma_f(d^\sharp)$.*

We can in particular take $\phi$ to be the formula defining the input-output relationships of a program in $L$ (that is, with real variables without loops), following the constructs in §3.2. By using 2.2 we can state:

**Corollary 2.** *Let $(D^\sharp_f, \alpha_f, \gamma_f)$ be the polynomial constraint domain defined by a finite family of polynomials $(f_\lambda)_{\lambda \in \Lambda}$. There is an algorithm that computes, given $P$ a program in $L$, and an element $d^\sharp \in D^\sharp_f$ with rational coefficients, rational bounds on the coefficients of the optimal approximation $\alpha_f \circ \overrightarrow{[\![P]\!]} \circ \gamma_f(d^\sharp)$, with arbitrary precision. The same holds with $\overleftarrow{[\![P]\!]}$ or if the coefficients of $d^\sharp$ are algebraic numbers defined by real closed field formulas.*

While we can obtain rational *bounds*, it is possible that the optimal coefficients are irrational: the optimal output interval of `if(x <0 || x*x >= 2) { x=0; }` is $[0, \sqrt{2}]$.

### 4.3 Optimal Abstract Fixpoints

We shall now show that we can also derive a relationship, expressed as a formula in the theory of real closed fields, between the parameters of an abstract state $s^\sharp$ and the least fixed point of the abstract semantics of a program fragment greater than $s^\sharp$. This gives a formula linking the parameters of the abstraction of the precondition of a loop to the parameters of an output abstract postcondition.

Let $\phi$ be a pre-post formula (over $V \cup V'$) and let $s^\sharp \in D^\sharp_f$. We are interested in the least fixed point (in $D^\sharp_f$) of $\alpha_f \circ \overrightarrow{\phi} \circ \gamma_f$. We shall show that it is possible to characterize such a fixed point using a formula in the theory of real closed fields. This means that for any polynomial constraint abstract domain, and any formula (for instance, a formula expressing the semantics of a program), the least fixed point in the abstract domain can be effectively computed.

In order to analyze program loops and similar constructs, we are interested in the strongest invariant containing some set $z_0$; an invariant of a monotonic function $f$ is a post-fixed point, that is, $z$ such that $f(z) \sqsubseteq z$. We recall the following result, similar to Tarski's fixed point theorem:

**Lemma 2.** *Let $(Z, \sqsubseteq, \sqcup, \sqcap)$ be a complete lattice and $z_0 \in Z$. Let $\psi : Z \to Z$ be an order-preserving operator. Then $\psi$ has a least post-fixed point above $z_0$, noted $\mathit{lpfp}_{z_0}\, \psi$; and this least post fixed point is* $\inf\{z \in Z \mid z_0 \sqsubseteq z \wedge f(z) \sqsubseteq z\}$. *$\mathit{lpfp}_\perp\, f$ is the least fixed point of $f$.*

It is possible to define $\sqsubseteq$ using a formula such that $d^\sharp \sqsubseteq d'^\sharp \iff (d^\sharp, d'^\sharp) \models \mathit{incl}(d^\sharp, d'^\sharp)$ where:

$$\mathit{incl}(d^\sharp, d'^\sharp) \triangleq d_b = 1 \vee (d'_b = 0 \wedge \mathit{lessEq}(d_1, d'_1) \wedge \cdots \wedge \mathit{lessEq}(d_m, d'_m)) \quad (8)$$

$$\mathit{less}(d, d') \triangleq \bar{d}' = 1 \vee (\bar{d}' = 0 \wedge \bar{d} = 0 \wedge d \leq d') \quad (9)$$

Now define:

$$\mathit{isFix}(d^\sharp, \psi^\sharp, d^\sharp_0) \triangleq \mathit{incl}(d^\sharp_0, d^\sharp) \wedge \psi^\sharp(d^\sharp, d^\sharp)$$

$$\mathit{isLfp}(d^\sharp, \psi^\sharp, d^\sharp_0) \triangleq \mathit{isFix}(d^\sharp, \psi^\sharp, d^\sharp_0) \wedge \forall d''^\sharp\ \mathit{isFix}(d''^\sharp, \psi^\sharp, d^\sharp_0) \Rightarrow \mathit{incl}(d^\sharp, d''^\sharp)$$

From these definitions, the following holds:

**Theorem 3.** *Let* $\psi^\sharp$ *be a formula over the variables* $d^\sharp = (d_b, (d_\lambda)_{\lambda\in\Lambda}, (\bar{d}_\lambda)_{\lambda\in\Lambda})$ *and* ${d'}^\sharp = (d'_b, (d'_\lambda)_{\lambda\in\Lambda}, (\bar{d}'_\lambda)_{\lambda\in\Lambda})$*, such that* $(d^\sharp, {d'}^\sharp) \models \psi^\sharp$ *defines an order-preserving function* $\overrightarrow{\psi^\sharp} : d^\sharp \mapsto {d'}^\sharp$*. Then, isLfp*$(d^\sharp, \psi^\sharp, d_0^\sharp)$ *has models* $(d^\sharp, d_0^\sharp) \models$ *isLfp*$(d^\sharp, \psi^\sharp, d_0^\sharp)$ *such that* $d^\sharp$ *is the least fixed point of* $\overrightarrow{\psi^\sharp}$ *over* $d_0^\sharp$*.*

By taking $\psi^\sharp = \mathit{abstrStep}(\phi, d^\sharp, {d'}^\sharp)$ and applying theorem 2:

**Corollary 3.** *Let* $\phi$ *be an input-output formula over variables* $V$*. Then, the constructed formula isLfp*$(d^\sharp, \mathit{abstrStep}(d^\sharp, \psi, d'^\sharp), o^\sharp)$ *has models* $(d^\sharp, o^\sharp)$*, where* $d^\sharp = (d_b, (d_\lambda)_{\lambda\in\Lambda}, (\bar{d}_\lambda)_{\lambda\in\Lambda})$ *and* $o^\sharp = (o_b, (o_\lambda)_{\lambda\in\Lambda}, (\bar{o}_\lambda)_{\lambda\in\Lambda})$*, exactly such that* $d^\sharp = \mathit{lpfp}_{o^\sharp}(\alpha_f \circ \overrightarrow{\phi} \circ \gamma_f)$*.*

The application to program analysis is that *fixpoints in the abstract domain may be approximated optimally, without the use of any widening operator*:

**Corollary 4.** *Let* $(D_f^\sharp, \alpha_f, \gamma_f)$ *be the polynomial constraint domain defined by a finite family of polynomials* $(f_\lambda)_{\lambda\in\Lambda}$*. There is an algorithm that computes, given* $P$ *a program in* $L$*, and an element* $s^\sharp \in D_f^\sharp$ *with rational coefficients, bounds on the coefficients of the optimal approximation* $\mathit{lpfp}_{s^\sharp}(\alpha_f \circ \overrightarrow{[\![P]\!]} \circ \gamma_f)$*, with arbitrary precision. The same holds with* $\overleftarrow{[\![P]\!]}$ *and/or if the coefficients of* $s^\sharp$ *are algebraic numbers defined by real closed field formulas.*

In order to compute an abstraction of the postcondition of a program `while` (*condition*) `{` *block* `}` given an abstraction $s^\sharp$ of the precondition, one can compute an abstraction of the reachable states at the head of the loop, and filter by $\neg$*condition*. The set of reachable states at the head of the loop is the least fixpoint of $X \mapsto [\![\mathit{block}]\!](X \cap [\![\mathit{condition}]\!])$ greater than $\gamma(s^\sharp)$, and an abstraction of this set is sought as a post-fixpoint of $[\![\mathit{condition}; \mathit{block}]\!]^\sharp$ greater than $s^\sharp$. Generally, this post-fixpoint is obtained using a *widening operator* [5, §4.3]: a sequence of candidates $s_1^\sharp, s_2^\sharp, \ldots$ is tried for being post-fixpoints, with a guarantee of termination; however, there is no guarantee of optimality. The above corollary gives an optimal characterization of the least fixed point $d^\sharp$ of $[\![\mathit{condition}; \mathit{block}]\!]^\sharp$ above $s^\sharp$, through formulas linking the coefficients of $d^\sharp$ to those of $s^\sharp$. Using the algorithms in §2.2, we can obtain bounds on the coefficients of this least fixed point, with arbitrary precision.

Note, however, that the optimal result that we obtain is the least fixed point *within the abstract domain*. In general, it is not the most precise abstraction of the concrete least fixed point. Computing the most precise abstraction of least fixed point would entail being able to solve the halting problem, and since the programs over the reals include the programs over the integers, this is impossible.

The difference between the two is as follows: instead of computing the abstraction $\alpha_f(\mathrm{lpfp}\,\psi)$ of the least fixed point of an operator, we compute $\mathrm{lpfp}\,\psi^\sharp$ the least fixed point of the optimal abstraction of that operator, $\psi^\sharp = \alpha_f \circ \psi \circ \gamma_f$. If $\psi$ is the semantics of a program fragment without loops, then $\psi$ is additive (the image of a union of sets is the union of the images of these sets); so is $\alpha_f$ (the image of a union of sets is the least upper bound of the images of these sets). Then, $\operatorname{lpfp} \psi = \bigcup_n \psi^n(\emptyset)$ [5, §4.1] and $\alpha_f(\operatorname{lpfp} \psi) = \alpha_f \left(\bigsqcup_n \psi^n(\emptyset)\right) = \bigsqcup_n \alpha_f \circ \psi^n \circ \gamma_f(\bot)$. In comparison, $\operatorname{lpfp} \psi^\sharp$ is $\bigsqcup_n (\alpha_f \circ \psi \circ \gamma_f)^n(\bot)$. Note that the two are identical if $\gamma_f \circ \alpha_f \circ \psi \circ \gamma_f = \psi \circ \gamma_f$. $\gamma_f \circ \alpha_f$ is a *upper closure operator* that maps each set $W \subseteq \mathbb{R}^V$ to the least superset representable in the abstract domain. This means that the whole loss of precision (difference between $\alpha_f(\operatorname{lpfp} \psi)$ and $\operatorname{lpfp} \psi^\sharp$) is caused by the loss of precision introduced by this closure.

### 4.4 Abstracting Programs with Loops

In section 4.2, we have shown how to effectively compute a family of optimal relations, which we shall note $\llbracket P \rrbracket_f^{\rightarrow\sharp}$, between the coefficients of an abstract value $d^\sharp$ in a polynomial constraint domain and those of the optimal abstraction of the postcondition, $\alpha_f \circ \overrightarrow{\llbracket P \rrbracket} \circ \gamma_f$.

In section 4.3, we have shown how to effectively compute a family of optimal relations between the coefficients of an abstract value $d^\sharp$ in a polynomial constraint domain and those of the least fixpoint of $\alpha_f \circ \overrightarrow{\llbracket \mathit{condition}; \mathit{block} \rrbracket} \circ \gamma_f$ greater than $d^\sharp$.

We can thus define a modular, compositional, forward abstract semantics $\llbracket P \rrbracket_f^{\rightarrow\sharp}$ by induction on the structure of the program. Given $P$, this semantics yields a family of formulas in the theory of real closed fields, linking the parameters of an abstract precondition $d^\sharp$ in $D_f^\sharp$ and the parameters of an abstract postcondition $d^{\sharp\prime}$ in $D_f^\sharp$ such that each of these parameters is uniquely defined. By using the algorithms given in section 2.2, these relationships can be used to compute the parameters in $d^{\sharp\prime}$ to arbitrary precision.

## 5 Related Works and Conclusion

We have defined an optimally precise abstract domain for programs without loops, based on formulas within the theory of real closed fields. This abstract domain can be used to derive optimal abstract transformers for a wide class of other domains, including familiar ones such as the interval and octagons domains. In addition, it can be also be used to provide optimal fixed points within those domains. The symbolic results that are computed can be algorithmically bounded with arbitrary precision, after applying quantifier elimination.

We have therefore demonstrated that, for a class of domains, widening operators are not needed in order to compute invariants. Moreover, our method, contrary to widenings, produces invariants that are optimal in a certain sense (the least invariant verifiable by the abstract transfer function).

There have been several published approaches to finding nonlinear relationships between program variables. One approach obtains polynomial equalities through computations on ideals using Gröbner bases [11]. This work only deals with equalities (not inequalities), uses a classical approach of computing output constraints from a set of input constraints (instead of finding relationships between the two sets of constraints), and deals with loops using a widening operator. In comparison, our approach abstracts whole program fragments, and is modular — it is possible to "plug" the result of the analysis of a procedure at the location of a procedure call.

Kapur, in some other work [6], proposes to use quantifier elimination to obtain invariants: he considers program invariants with parameters, and derives constraints over those parameters from the program.[11] Our work improves on his by noting that least invariants of the chosen shape can be obtained, not just any invariant; that the abstraction can be done modularly and compositionally (a program fragment can be analyzed, and the result of its analysis can be plugged into the analysis of a larger program), or combined into a "conventional" abstract interpretation framework (by using invariants of a shape compatible with that framework), and that the resulting invariants can be "projected" to obtain numerical quantities.

There have been other methods proposed for generating invariants from fixed parametric "shapes", using constraints over the parameters. Some approaches apply numerical algorithms, such as linear programming [13] or other forms of constraint solving. One difference with our work is that such methods will solve the invariant problem for *one* set of numerical values for input constraints, while we provide formulas that are valid for all sets of inputs (and then, that can be instanciated as many times as necessary to obtain numerical invariants). Some of the proposed techniques consider non-linear invariants; for instance, some [12,10] find coefficients for algebraic *equalities* ($P(v_1, \dots, v_n) = 0$), using techniques of Gröbner bases. Our technique finds optimal algebraic *inequalities*, and, furthermore, obtains constraints linking their parameters to constraints on program inputs.

Quantifier elimination in the theory of real closed fields is a very costly operation; thus, our algorithms, taken "as is", are likely not to be tractable beyond simple cases. However, from a theoretical point of view, it is interesting to note that widening operators are not needed in order to guarantee the computability of least fixed points in e.g. the real interval domain. We also hope that "approximate" quantifier elimination techniques (providing $Q$ such that $\exists x\ P \implies Q$, instead of $Q$ such that $\exists x\ P \iff Q$) may make some of our algorithms more tractable. Some experiments suggest that the algorithms can be made more efficient in the linear case, using geometric techniques, and we hope to provide more results in that respect.

With respect to applications, we envision the automatic synthesis of transfer functions for "conventional" abstract interpreters. One limitation of systems such as Astrée [2] is that, for each program construct, and each abstract domain, an abstract transfer function must be programmed by hand. If one feels like a whole block of instructions should be analyzed as a whole, in order to get more precision, then one has to derive the necessary transfer function and implement it, with risks of introducing bugs. We think that techniques such as the one in

[11] We thank Enea Zaffanella for pointing out this work to us.

this paper, or improvements thereof, could be used to provide generic transfer functions, possibly through dynamic code generation.

## References

1. Basu, S., Pollack, R., Roy, M.-F.: Algorithms in real algebraic geometry. In: Algorithms and computation in mathematics, Springer, Heidelberg (2003)
2. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: Mogensen, T., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation, vol. 2566. LNCS, pages 85–108. Springer, Heidelberg 2002.
3. Caviness, B.F., Johnson, J.R. (eds.): Quantifier elimination and cylindrical algebraic decomposition. Springer, Heidelberg (1998)
4. Clarisó, R., Cortadella, J.: The octahedron abstract domain. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, Springer, Heidelberg (2004)
5. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. Journal of Logic Programming 13(2–3), 103–179 (1992), A correctly typeset version. See `http://www.di.ens.fr/~cousot`
6. Kapur, D.: Automatically generating loop invariants using quantifier elimination. In: ACA (Applications of Computer Algebra) (2004)
7. Lang, S.: Algebra, 3rd edn. Addison-Wesley, Reading (1993)
8. Miné, A.: The octagon abstract domain. In: Simon, A., King, A., Howe, J. (eds.) WCRE, Analysis, Slicing, and Transformation, pp. 310–319. IEEE, Los Alamitos (2001)
9. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, Springer, Heidelberg (2004)
10. Rodríguez-Carbonell, E., Kapur, D.: Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In: International Symposium on Symbolic and Algebraic Computation 2004 (ISSAC04), pp. 266–273. ACM Press, New York (2004)
11. Rodríguez-Carbonell, E., Kapur, D.: An abstract interpretation approach for automatic generation of polynomial invariants. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, Springer, Heidelberg (2004)
12. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Non-linear loop invariant generation using gröbner bases. In: POPL, ACM, New York (2004)
13. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 21–47. Springer, Heidelberg (2005)
14. Turing, A.: On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, series 2, 42, 230–265 (1936)
15. Weihrauch, K.: Computable analysis: an introduction. In: Texts in Theoretical Computer Science, Springer, Heidelberg (2000)
16. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. In: Foundations of Computing, MIT Press, Cambridge (1993)

# Taming the Wrapping of Integer Arithmetic

Axel Simon and Andy King

Computing Laboratory, University of Kent, Canterbury, UK
{a.simon,a.m.king}@kent.ac.uk

**Abstract.** Variables in programs are usually confined to a fixed number of bits and results that require more bits are truncated. Due to the use of 32-bit and 64-bit variables, inadvertent overflows are rare. However, a sound static analysis must reason about overflowing calculations and conversions between unsigned and signed integers; the latter remaining a common source of subtle programming errors. Rather than polluting an analysis with the low-level details of modelling two's complement wrapping behaviour, this paper presents a computationally light-weight solution based on polyhedral analysis which eliminates the need to check for wrapping when evaluating most (particularly linear) assignments.

## 1 Introduction

Static analysis methods are increasingly used to prove the partial correctness of software [5]. In contrast to formal methods that verify properties of a high-level specification, a static analysis is complicated by low-level details of source code. For instance, while a specification expresses properties over arbitrary integers, variables in a program are usually confined to finite integer types that are deemed to be large enough to hold all values occurring at run-time. On one hand, the use of 32-bit and 64-bit variables make accidental overflows rare and adding checks to each transfer function of the analysis seems to be excessive considering the infrequency of variable overflows. On the other hand, programmers often inadvertently introduce wrapping when converting between signed and unsigned variables and deliberately exploit the wrapping effects of two's complement arithmetic. Thus, wrapping itself should not be considered harmful, particularly when the objective of an analysis is the verification of a different property, such as absence of out-of-bound memory accesses [8,9].

In fact, there is a danger in flagging all wrapping, since any intentional use of wrapping generates a warning message which the developer immediately dismisses as a false positive. The code in Figure 1 illustrates why this is a problem. The purpose of the shown C function is to print how many times each individual character occurs in the given string `*str`. To this end, the elements of `dist` are initialised to zero by the call to `memset`. In the loop that follows, the $n$th element is incremented each time a character $n$ is encountered in `str`. The `for` loop then prints the distribution of printable ASCII characters.

The shown program is correct on platforms where `char` is unsigned such as Linux on PowerPC. However, for Linux on x86 and MacOS X on PowerPC, `char`

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 121–136, 2007.

```
void showDistribution(char* str) {
  int i;
  int dist[256];             /* Table of character counts.*/

  memset(dist, 0, sizeof(dist));            /* Clear table.*/

  while (*str) {
    dist[(unsigned int) *str]++;
    str++;
  };

  for(i=32; i<128; i++)        /* Show dist for printable */
    printf("'%c'␣:␣%i\n", i, dist[i]);      /* characters.*/
}
```

**Fig. 1.** Example C function that counts the occurrences of each character

is signed. In the latter case, the value `*str` which is used to index into `dist` can take on values in the range $[-128, 127]$. Although the programmer intended to convert the value of `*str` to an unsigned value before the extension to a 4-byte quantity takes place, the C standard [3] dictates that the value of `*str` is first promoted to `int` before the conversion to an unsigned type is performed. Hence, `dist` can be accessed at indices $[2^{32}-128 \ldots 2^{32}-1] \cup [0 \ldots 127]$, of which the first range is out-of-bounds. A static analysis that considers all wrapping to be erroneous would flag this statement as possibly faulty. However, since the programmer expects that wrapping does occur (namely when converting from a `char` to an unsigned quantity), the warning about wrapping at the `unsigned int`-level will be dismissed as a false positive. Hence, the analysis above should flag the out-of-bound array access but treat wrapping itself as intentional.

In this work, we propose a re-interpretation of polyhedra in which the wrapping of integer calculations is reflected in the classic polyhedral domain [6]. In particular, we avoid making cross-cutting changes to all transfer functions but refine the approximation relation such that wrapping is mostly implicit, that is, the need for extra polyhedral operations is largely finessed. For the few cases in which wrapping has to be addressed in the transfer functions, we illustrate how to wrap values within the polyhedral domain and propose an algorithm for doing so. To summarise our contributions, this paper presents:

– an approximation relation that implicitly wraps polyhedral variables;
– an algorithm to perform wrapping in the polyhedral domain;
– a formal description of an analysis that faithfully models wrapping and an accompanying correctness argument.

We commence with the definition of a small language and its concrete semantics. Section 3 introduces polyhedral analysis. Sections 4 and 5 explain how wrapping is supported. Section 6 presents a wrapping-aware polyhedral analysis which is discussed in Section 7. We conclude with the related work in Section 8.

## 2 A Language Featuring Finite Integer Arithmetic

The function shown in Figure 1 demonstrates that it is important to clarify where wrapping can arise in a program. This is particularly true when arguing the correctness of an analysis. To this end we introduce the language $\mathcal{L}$(ELang) which is a subset of an intermediate language which is used to analyse C programs.

### 2.1 The Syntax of $\mathcal{L}$(ELang)

$\mathcal{L}$(ELang) features linear expressions and casts between integers. In the following grammar of $\mathcal{L}$(ELang), (T)* denotes the repetition of the non-terminal T.

$$\begin{array}{ll}
\langle \text{ELang} \rangle & :: (\text{Block})^* \\
\langle \text{Block} \rangle & :: l \,\mathbf{:}\, (\langle \text{Stmt} \rangle \,\mathbf{;})^* \, \langle \text{Next} \rangle \\
\langle \text{Next} \rangle & :: \mathbf{jump}\ l\,\mathbf{;} \\
 & \mid\ \mathbf{if}\ \langle \text{Type} \rangle\ v\ \langle \text{Op} \rangle\ \langle \text{Expr} \rangle\ \mathbf{then\ jump}\ l\,\mathbf{;}\ \langle \text{Next} \rangle \\
\langle \text{Op} \rangle & :: < \mid \leq \mid = \mid \neq \mid \geq \mid > \\
\langle \text{Expr} \rangle & :: n \mid n\ \mathbf{*}\ v\ \mathbf{+}\ \langle \text{Expr} \rangle \\
\langle \text{Stmt} \rangle & :: \langle \text{Size} \rangle\ v\ \mathbf{=}\ \langle \text{Expr} \rangle \\
 & \mid\ \langle \text{Size} \rangle\ v\ \mathbf{=}\ \langle \text{Type} \rangle\ v \\
\langle \text{Type} \rangle & :: (\mathbf{uint} \mid \mathbf{int})\ \langle \text{Size} \rangle \\
\langle \text{Size} \rangle & :: \mathbf{1} \mid \mathbf{2} \mid \mathbf{4} \mid \mathbf{8}
\end{array}$$

An ELang program consists of a sequence of basic blocks with execution commencing with the first block. Each basic block consists of a sequence of statements and a list of control-flow instructions. In the sequel, we use *lookupBlock*$(l)$ and *lookupNext*$(l)$ to access the statements and control-flow instructions, respectively, of basic block $l$. The control-flow of a basic block consists of either a **jump** statement or a conditional which itself is followed by more control flow instructions. The $\langle$Stmt$\rangle$ production is restricted to the two statements of interest, namely the assignment of linear expressions to a variable and a type cast. We require that each variable in the program is used with only one size which is always specified in bytes. In particular, the assignment statement and the conditional require that all occurring variables are of the same size. Note, though, that variables may be used as an **uint** (unsigned integer) in one statement and as an **int** (signed integer) in another.

### 2.2 The Semantics of $\mathcal{L}$(ELang)

In order to specify the semantics of $\mathcal{L}$(ELang), we introduce the following notation. Let $\mathbb{B} = \{0, 1\}$ denote the set of Booleans. A vector $\boldsymbol{b} = \langle b_{w-1}, \ldots b_0 \rangle \in \mathbb{B}^w$ is interpreted as unsigned integer by $val^{w,\text{uint}}(\boldsymbol{b}) = \sum_{i=0}^{w-1} b_i 2^i$ and as signed integer by $val^{w,\text{int}}(\boldsymbol{b}) = (\sum_{i=0}^{w-2} b_i 2^i) - b_{w-1} 2^{w-1}$. Conversely, $bin^w : \mathbb{Z} \to \mathbb{B}^w$ converts an integer to the lower $w$ bits of its Boolean representation. Formally, $bin^w(v) = \boldsymbol{b}$ iff there exists $\boldsymbol{b}' \in \mathbb{B}^q$ such that $val^{(q+w),\text{int}}(\boldsymbol{b}' \| \boldsymbol{b}) = v$ where $\|$ denotes the concatenation of bit-vectors. For instance, $bin^3(15) = \langle 1, 1, 1 \rangle$ since

Basic Blocks.

$$\llbracket\, l : s_1; \ldots s_n;\, \rrbracket^\natural_{\text{Block}}\sigma = \llbracket\, \mathit{lookupNext}(l) \,\rrbracket^\natural_{\text{Next}}(\llbracket\, s_n \,\rrbracket^\natural_{\text{Stmt}}(\ldots(\llbracket\, s_1 \,\rrbracket^\natural_{\text{Stmt}}\sigma)\ldots))$$

Control Flow.

$$\llbracket\, \mathbf{jump}\ l \,\rrbracket^\natural_{\text{Next}}\sigma = \llbracket\, \mathit{lookupBlock}(l) \,\rrbracket^\natural_{\text{Block}}\sigma$$

$$\llbracket\, \mathbf{if}\ t\ s\ v\ \mathit{op}\ \mathit{exp}\ \mathbf{then\ jump}\ l\ ;\ \mathit{nxt} \,\rrbracket^\natural_{\text{Next}}\sigma = \begin{cases} \llbracket\, \mathit{lookupBlock}(l) \,\rrbracket^\natural_{\text{Block}}\sigma & \text{if } \mathit{val}^{8s,t}(\sigma^s(\mathit{addr}^\natural(v)))\ \mathit{op}\ \mathit{val}^{8s,t}(\llbracket\, \mathit{exp} \,\rrbracket^{\natural,s}_{\text{Expr}}\sigma) \\ \llbracket\, \mathit{nxt} \,\rrbracket^\natural_{\text{Next}}\sigma & \text{otherwise} \end{cases}$$

Expressions.

$$\llbracket\, n \,\rrbracket^{\natural,s}_{\text{Expr}}\sigma = \mathit{bin}^{8s}(n)$$

$$\llbracket\, n * v + \mathit{exp} \,\rrbracket^{\natural,s}_{\text{Expr}}\sigma = \mathit{bin}^{8s}(n) *^{8s} \sigma^s(\mathit{addr}^\natural(v)) +^{8s} \llbracket\, \mathit{exp} \,\rrbracket^{\natural,s}_{\text{Expr}}\sigma$$

Assignment.

$$\llbracket\, s\ v = \mathit{exp} \,\rrbracket^\natural_{\text{Stmt}}\sigma = \sigma[\mathit{addr}^\natural(v) \overset{s}{\mapsto} \llbracket\, \mathit{exp} \,\rrbracket^{\natural,s}_{\text{Expr}}\sigma]$$

Type Casts.

$$\llbracket\, s_1\ v_1 = t\ s_2\ v_2 \,\rrbracket^\natural_{\text{Stmt}}\sigma = \sigma[\mathit{addr}^\natural(v_1) \overset{s_1}{\mapsto} \mathit{bin}^{8s_1}(\mathit{val}^{8s_2,t}(\sigma^{s_2}(\mathit{addr}^\natural(v_2))))]$$

**Fig. 2.** Concrete Semantics of $\mathcal{L}$(ELang)

$\mathit{val}^{5,\text{int}}(\langle 0,1,1,1,1\rangle) = 15$. In order to distinguish calculations on Boolean vectors from standard arithmetic, let $+^w, *^w : \mathbb{B}^w \times \mathbb{B}^w \to \mathbb{B}^w$ denote addition and multiplication that truncate the result to the lower $w$ bits, for instance $\langle 1,1,1,1\rangle +^4 \langle 0,0,0,1\rangle = \langle 0,0,0,0\rangle$. Note that the signedness of the arguments of $+^w$ and $*^w$ do not affect the result of these operations.

$\mathcal{L}$(ELang) programs operate in a virtual memory environment which we formalise as a sequence of bytes. Let $\mathcal{B} = \mathbb{B}^8$ denote the set of bytes and $\Sigma = \mathcal{B}^{2^{32}}$ all states of 4 GByte that a program on a 32-bit architecture can take on. Let $\sigma \in \Sigma$ denote a given memory state of a program and let $\sigma^s : [0, 2^{32} - 1] \to \mathcal{B}^s$ denote a read access at the given 32-bit address where $s \in \{1, 2, 4, 8\}$ is the number of bytes to be read. A write operation is formalised as a substitution $\sigma[a \overset{s}{\mapsto} v]$. The resulting store $\sigma' = \sigma[a \overset{s}{\mapsto} v]$ satisfies $\sigma'^s(a) = v$ and furthermore $\sigma'^1(b) = \sigma^1(b)$ for all addresses $b \notin \{a, \ldots a + s - 1\}$.

Figure 2 presents the concrete semantics (or natural semantics, hence the $\natural$) of the $\mathcal{L}$(ELang) language. These definitions use $\mathit{addr}^\natural(v) \in [0, 2^{32} - 1]$ which maps the program variable $v$ to its address in memory. We assume that $\mathit{addr}^\natural$ maps different variables to non-overlapping memory regions, an assumption that makes $\mathcal{L}$(ELang) independent of the endianness of an architecture.

The concrete semantics manipulates the store mainly by operations on bit-vectors; only in the conditional and in the cast are bit vectors interpreted as numbers. In these cases the signedness of the variables can actually influence the result. In particular, the type $t$ of the cast determines if the source bit-vector is sign-extended (if $t = \text{int}$) or zero-extended (if $t = \text{uint}$) when $s_1 \geq s_2$. We now proceed by abstracting this semantics so as to specify a polyhedral analysis.

# 3 Polyhedral Analysis of Finite Integers

Two's complement arithmetic exploits the wrapping behaviour of integer variables that are confined to a fixed number of bits. For instance, subtracting 1 from an integer is equivalent to adding the largest representable integer value. In fact, the binary representation of the signed integer $-1$ is identical to that of the largest, unsigned integer of the same size. In the context of verification, this dichotomy in interpretation cannot be dismissed since $\mathcal{L}(\text{ELang})$ has insufficient information about the signedness of assignments. This omission allows our model to be applied to languages which freely mix signed and unsigned values, e.g. C.

Accessing the same bit sequence as either signed or unsigned integer corresponds to a wrapping behaviour in that the negative range of the signed integer wraps to the upper range of an unsigned integer, see Figure 3. This wrapping behaviour of finite integers creates a mismatch against the infinite range of polyhedral variables. We present our solution to this mismatch in two parts: Section 4 presents a concretisation map between the polyhedral domain and the bit-level representation of variables. This map wraps values of abstract variables implicitly to finite sequences of bits, thereby alleviating the need to check for wrapped values each time a variable is read or written. In contrast, Section 5 details an algorithm that makes the wrapping of program variables explicit in the abstract domain which is important for casts and the conditional statement whose semantics depend on the size and signedness of the operands.

## 3.1 The Domain of Closed, Convex Polyhedra

In order to make the paper self-contained, this section gives a concise introduction to the notation used in our polyhedral analysis. Let $X = \{x_1, \ldots x_n\}$ be a finite set of variables, let $\boldsymbol{x} = \langle x_1, \ldots x_n \rangle$ and let $I$ be the set of linear inequalities that can be rewritten as $\boldsymbol{c} \cdot \boldsymbol{x} \leq d$ where $\boldsymbol{c} \in \mathbb{Z}^n$ and $d \in \mathbb{Z}$. For brevity we write $e_1 = e_2$ to denote the set of inequalities $\{e_1 \leq e_2, e_2 \leq e_1\}$ where $e_i$ is any linear expression. Furthermore, let $e_1 < e_2$ abbreviate $e_1 \leq e_2 - 1$. Let $\iota \in I$ be an inequality that can be transformed into $\boldsymbol{c} \cdot \boldsymbol{x} \leq d$, then $[\![\iota]\!] = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{c} \cdot \boldsymbol{x} \leq d\}$ denotes the induced half-space. Given a finite set of inequalities $E = \{\iota_1, \ldots \iota_n\} \subseteq I$ the notation $[\![E]\!] = \bigcap_{i=1,\ldots n} [\![\iota_i]\!]$ denotes the induced convex polyhedron. Let

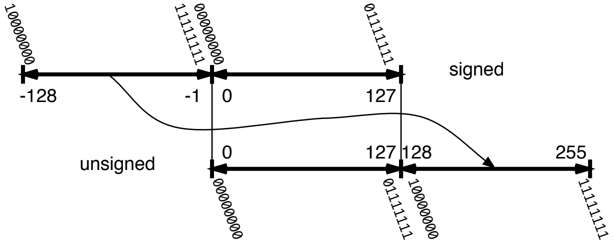


**Fig. 3.** The difference between a signed and an unsigned access can be interpreted as a wrap of negative values to the upper range in an unsigned access

*Poly* be the set of all convex polyhedra (polyhedra for short). Given two polyhedra $P_1, P_2 \in \mathit{Poly}$, let $P_1 \sqsubseteq P_2$ iff $P_1 \subseteq P_2$ and let $P_1 \sqcap P_2 = P_1 \cap P_2$. Note that $P_1 \sqcap P_2 = [\![E_1 \cup E_2]\!]$ where $P_i = [\![E_i]\!]$ and $i = 1, 2$. We write $P_1 \sqcup P_2$ to denote the closure of the convex hull [6] of two polyhedra $P_1$ and $P_2$ which is defined as the smallest polyhedron $P$ such that $P_1 \cup P_2 \sqsubseteq P$. Let $P \in \mathit{Poly}$ be non-empty and $V_i = \{v_i \mid \langle v_0, \ldots v_n \rangle \in P\}$. We write $P(x) = [l, u]$ where $l = \lceil \min(V_i) \rceil$ if the minimum exists, otherwise $l = -\infty$ and $u = \lfloor \max(V_i) \rfloor$ if the maximum exists, otherwise $u = \infty$. Define $\exists_{x_i} : \mathit{Poly} \to \mathit{Poly}$ such that $\exists_{x_i}(P) = \{\langle v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots v_n \rangle | \langle v_1, \ldots v_n \rangle \in P \wedge x \in \mathbb{R}\}$. Let $T$ denote an additional set of temporary variables such that $T \cap X = \emptyset$. All operations above lift from $X$ to $T \cup X$. The operation $P \triangleright x := e \in \mathit{Poly}$ denotes an update of a variable $x \in X$ to the linear expression $e$ and is defined as $\exists_t([\![x = t]\!] \sqcap \exists_x([\![t = e]\!] \sqcap P))$ where $t \in T$ does not occur in $e$ and $t \neq x$. The rationale for assigning the result to a temporary variable is that $e$ might contain $x$. Assigning $e$ to a fresh variable $t$ will retain the value of $x$ before it is projected out. If $x$ does not occur in $e$, the update above is equivalent to $\exists_x(P) \sqcap [\![x = e]\!]$. In order to argue about updates of polyhedra, observe the following:

**Lemma 1.** *Let $P \in \mathit{Poly}$ and $P' = P \triangleright x_i := \boldsymbol{c} \cdot \boldsymbol{x} + d$ with $\boldsymbol{c} \in \mathbb{Z}^n$ and $d \in \mathbb{Z}$. Then $P' = \{\langle v_1, \ldots v_{i-1}, v'_i, v_{i+1}, \ldots v_n \rangle \mid \boldsymbol{v} = \langle v_1, \ldots v_n \rangle \in P \wedge v'_i = \boldsymbol{c} \cdot \boldsymbol{v} + d\}$.*

## 4 Implicit Wrapping of Polyhedral Variables

This section formalises the relationship between polyhedral variables and bit-sequences that constitute the program state. For simplicity, we assume a one-to-one correspondance between the variable names in the program and the polyhedral variables that represent their values. The values of a program variable are merely bit sequences that are prescribed by the possible values of the polyhedral variable. To illustrate, suppose that `x` is of type `char` and $P(x) = [-1, 2]$. The represented bit patterns are 11111111, 00000000, 00000001 and 00000010, no matter whether `x` is signed or unsigned. These bit patterns are given by $\mathit{bin}^{8s}(v)$ which turns a value $v \in [-1, 2]$ into a bit sequence of $s$ bytes. Going further, the function $\mathit{bits}^s_a : \mathbb{Z} \to \mathcal{P}(\Sigma)$ produces all concrete stores in which $8s$ bits at address $a = \mathit{addr}^\natural(\mathtt{x})$ are set to the value corresponding to $v \in P(x)$ as follows:

$$\mathit{bits}^s_a(v) = \{\langle r_{8*2^{32}} \ldots r_{8(a+s)} \rangle \| \, \mathit{bin}^{8s}(v) \, \| \, \langle r_{8a-1} \ldots r_0 \rangle \mid r_i \in \mathbb{B}\}$$

Note that this definition only considers the lower $8s$ bits of the value $v$. For instance, $\mathit{bits}^1_a(0) = \mathit{bits}^1_a(256)$ since the lower eight bits of 0 and 256 are equal. The mapping $\mathit{bits}^s_a$ can be lifted from the value $v$ of a single variable to the values $\langle v_1, \ldots v_n \rangle \in \mathbb{Z}^n$ of a vector of variables $\langle x_1, \ldots x_n \rangle$, resulting in the stores $\bigcap_{i \in [1,n]} \mathit{bits}^{s_i}_{a_i}(v_i)$. Here $a_i \in [0, 2^{32} - 1]$ denotes the address of the variable $x_i$ in the concrete store and $s_i \in \mathbb{N}$ denotes its size in bytes. Observe that, if variables were allowed to overlap, the above intersection might incorrectly collapse to the empty set for certain vectors $\langle v_1, \ldots v_n \rangle \in \mathbb{Z}^n$. Using this lifting, a polyhedron is now related to a set of stores by $\gamma^{\boldsymbol{s}}_{\boldsymbol{a}} : \mathit{Poly} \to \mathcal{P}(\Sigma)$ which is defined as

$$\gamma^{\boldsymbol{s}}_{\boldsymbol{a}}(P) = \bigcup_{\boldsymbol{v} \in P \cap \mathbb{Z}^n} \left( \bigcap_{i \in [1,n]} \mathit{bits}^{s_i}_{a_i}(v_i) \right)$$

where $\boldsymbol{s} = \langle s_1, \ldots s_n \rangle$, $\boldsymbol{a} = \langle a_1, \ldots a_n \rangle$ and $\boldsymbol{v} = \langle v_1, \ldots v_n \rangle$.

The definition of $\gamma^{\boldsymbol{s}}_{\boldsymbol{a}}$ provides a criterion for judging the correctness of an abstract semantics. In addition, $\gamma^{\boldsymbol{s}}_{\boldsymbol{a}}$ permits linear expressions to be evaluated in the abstract semantics without the need to address overflows since $\gamma^{\boldsymbol{s}}_{\boldsymbol{a}}$ maps the result of calculations in the polyhedral domain to the correctly wrapped result in the actual program. This property is formalised below:

**Proposition 1.** *Let $e \in \mathcal{L}(\mathit{Expr})$ and $e \equiv \boldsymbol{c} \cdot \boldsymbol{x} + d$, that is, $e$ is a reformulation of $\boldsymbol{c} \cdot \boldsymbol{x} + d$. If $\sigma \in \gamma^{\boldsymbol{s}}_{\boldsymbol{a}}(P)$ then $\sigma[a_i \overset{s_i}{\mapsto} [\![ e ]\!]^{\natural,s_i}_{\text{Expr}}] \in \gamma^{\boldsymbol{s}}_{\boldsymbol{a}}(P \rhd x_i := \boldsymbol{c} \cdot \boldsymbol{x} + d)$.*

*Proof.* Define $\pi_i(\langle x_1, \ldots x_{i-1}, x_i, x_{i+1}, \ldots x_n \rangle) = x_i$. Since $\sigma \in \gamma^{\boldsymbol{s}}_{\boldsymbol{a}}(P)$ there exists $\boldsymbol{v} \in P \cap \mathbb{Z}^n$ such that $\sigma = \bigcap_{i \in [1,n]} \mathit{bits}^{s_i}_{a_i}(\pi_i(\boldsymbol{v}))$. Let $P' = P \rhd x_i := \boldsymbol{c} \cdot \boldsymbol{x} + d$ for some $\boldsymbol{c} \in \mathbb{Z}^n$ and $d \in \mathbb{Z}$. By Lemma 1, there exists $\boldsymbol{v}' \in P'$ with $\pi_j(\boldsymbol{v}') = \pi_j(\boldsymbol{v})$ for all $j \neq i$. Since $\{a_i, \ldots a_i + s_i - 1\} \cap \{a_j, \ldots a_j + s_j - 1\} = \emptyset$ for all $j \neq i$, there exists $\sigma' \in \gamma^{\boldsymbol{s}}_{\boldsymbol{a}}(P')$ such that $\sigma'^1(a) = \sigma^1(a)$ for $a \in [0, 2^{32} - 1] \setminus \{a_i, \ldots a_i + s_i - 1\}$. Furthermore, the lemma states that $\pi_i(\boldsymbol{v}') = \boldsymbol{c} \cdot \boldsymbol{v} + d$ and, by the definition of $\gamma^{\boldsymbol{s}}_{\boldsymbol{a}}$, it follows that $\sigma'^{s_i}(a_i) = \mathit{bin}^{8s_i}(\boldsymbol{c} \cdot \boldsymbol{v} + d)$. To show that $\sigma'^{s_i}(a_i) = [\![ e ]\!]^{\natural,s_i}_{\text{Expr}}\sigma$, we find $\boldsymbol{a} \in \mathbb{Z}^n$, $d \in \mathbb{Z}$ such that $e \equiv \boldsymbol{c} \cdot \boldsymbol{x} + d$ and $[\![ e ]\!]^{\natural,s_i}_{\text{Expr}}\sigma = \mathit{bin}^{8s_i}(\boldsymbol{c} \cdot \boldsymbol{v} + d)$ by induction over $e$:

1. Let $e = n$. By definition of $[\![ \cdot ]\!]^{\natural,s}_{\text{Expr}}$, $[\![ n ]\!]^{\natural,s_i}_{\text{Expr}}\sigma = \mathit{bin}^{8s_i}(n) = \mathit{bin}^{8s_i}(\boldsymbol{c} \cdot \boldsymbol{v} + d)$ where $d = n$ and $\boldsymbol{c} = \langle 0, \ldots 0 \rangle$. Hence $e \equiv \boldsymbol{c} \cdot \boldsymbol{x} + d$.
2. Let $e = n * x_j + e'$. Suppose that $[\![ e' ]\!]^{\natural,s_i}_{\text{Expr}}\sigma = \mathit{bin}^{8s_i}(\boldsymbol{c}' \cdot \boldsymbol{v} + d')$ where $e' \equiv \boldsymbol{c}' \cdot \boldsymbol{x} + d'$. By the definition of $[\![ \cdot ]\!]^{\natural,s}_{\text{Expr}}$, $[\![ n * x_j + e ]\!]^{\natural,s_i}_{\text{Expr}}\sigma = \mathit{bin}^{8s}(n) *^{8s_i} \sigma^{s_i}(a_j) +^{8s_i} [\![ e' ]\!]^{\natural,s_i}_{\text{Expr}}\sigma$ where $\sigma^{s_i}(a_j) = \mathit{bin}^{8s_i}(v_j)$. By definition of $\mathit{bin}^{8s}$, $\mathit{bin}^{8s}(n) *^{8s_i} \mathit{bin}^{8s_i}(v_j) = ((n \mathbin{\mathit{mod}} 2^{8s_i}) * (v_j \mathbin{\mathit{mod}} 2^{8s_i})) \mathbin{\mathit{mod}} 2^{8s_i} = (n * v_j) \mathbin{\mathit{mod}} 2^{8s_i}$, c.f. [7, page 42]. Similarly, $(n * v_j) \mathbin{\mathit{mod}} 2^{8s_i} +^{8s_i} [\![ e' ]\!]^{\natural,s_i}_{\text{Expr}}\sigma = (n * v_j) \mathbin{\mathit{mod}} 2^{8s_i} +^{8s_i} \mathit{bin}^{8s_i}(\boldsymbol{c}' \cdot \boldsymbol{v} + d') = (n * v_j + \boldsymbol{c}' \cdot \boldsymbol{v} + d') \mathbin{\mathit{mod}} 2^{8s_i}$. Thus, set $d = d'$ and $\langle c_1, \ldots c_n \rangle = \langle c'_1, \ldots c'_{i-1}, c'_i + n, c'_{i+1}, \ldots c'_n \rangle$ where $\boldsymbol{c} = \langle c_1, \ldots c_n \rangle$ and $\boldsymbol{c}' = \langle c'_1, \ldots, c'_n \rangle$. Hence $e \equiv \boldsymbol{c} \cdot \boldsymbol{x} + d$.

The force of the above result is that a linear expression ⟨Expr⟩ over finite integer variables can be interpreted as an expression over polyhedral variables without regard for overflows or evaluation order. A prerequisite for this convenience is that all variables occurring in an expression have the same size $s$. In contrast, assignments between different sized variables have to revert to a cast statement. In this case, and in the case of conditionals, wrapping has to be made explicit which is the topic of the next section.

## 5 Explicit Wrapping of Polyhedral Variables

A consequence of the wrapping behaviour of $\gamma^{\boldsymbol{s}}_{\boldsymbol{a}}$ is that the effect of a guard such as `x<=y` cannot be modelled as a transformation from a polyhedron $P$ to

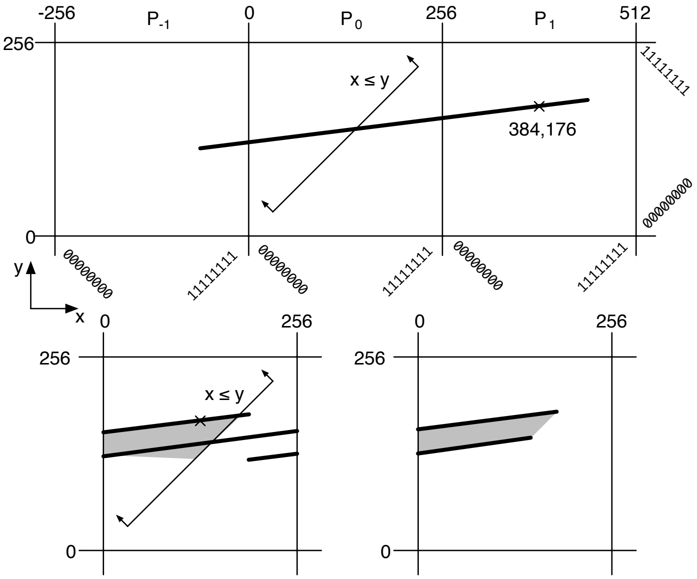


**Fig. 4.** Explicitly wrapping the possible values of $x$ to its admissible range

$P \sqcap [\![x \leq y]\!]$. This section explains this problem, discusses possible solutions and proposes an efficient wrapping algorithm *wrap*.

### 5.1 Wrapping Variables with a Finite Range

In order to illustrate the requirements on the *wrap* function, consider Figure 4. The thick line in the upper graph denotes $P = [\![x + 1024 = 8y, -64 \leq x \leq 448]\!]$ which we suppose feeds into the guard x<=y where x and y both represent variables of type **uint8**. In order to illustrate a peculiarity of modelling the guard, consider the point $\langle x, y\rangle = \langle 384, 176\rangle \in P$ and let $\sigma \in \gamma^{\boldsymbol{s}}_{\boldsymbol{a}}(\{\langle 384, 176\rangle\})$. Due to implicit wrapping in $\gamma^{\boldsymbol{s}}_{\boldsymbol{a}}$, the state $\sigma$ stipulates that $val^{8,\mathrm{uint}}(\sigma^1(addr^{\natural}(x))) = 128$ and $val^{8,\mathrm{uint}}(\sigma^1(addr^{\natural}(y))) = 176$. Thus, although x<=y is true when interpreting x and y as **uint8** in $\sigma$, the polyhedron $\{\langle 384, 176\rangle\} \sqcap [\![x \leq y]\!]$ is empty. Hence, it is not correct to model the guard in the classic way as $P \sqcap [\![x \leq y]\!]$.

In order to model relational tests correctly, the values of expressions occurring on each side of a relational operator have to be wrapped to the type prescribed in an $\mathcal{L}$(ELang) conditional. In the example, the expression $y$ is already in the required range $[0, 255]$ whereas the range of $x$ impinges on the two neighbouring quadrants as indicated in the upper graph of Figure 4. These quadrants are obtained by partitioning the state $P$ into $P_{-1} = P \sqcap [\![-256 \leq x \leq -1]\!]$, $P_0 = P \sqcap [\![0 \leq x \leq 255]\!]$ and $P_1 = P \sqcap [\![256 \leq x \leq 511]\!]$. The result of wrapping $x$ can now be calculated by translating $P_{-1}$ by 256 units towards positive $x$-coordinates and $P_1$ by 256 units towards negative $x$-coordinates, yielding $P' = P_0 \sqcup (P_{-1} \triangleright x := x + 256) \sqcup (P_1 \triangleright x := x - 256)$. The contribution of each

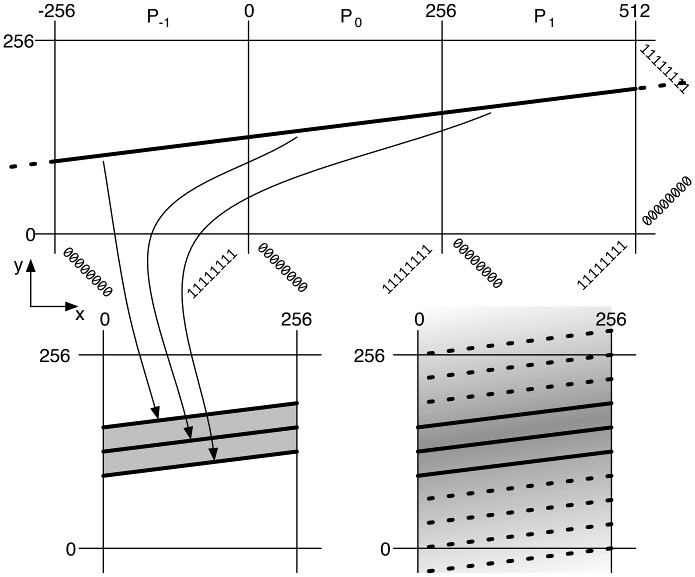


**Fig. 5.** The quest for an efficient wrapping of unbounded variables

partition is shown as a thick line in the lower left graph and the grey region depicts $P' \sqcap [\![x \leq y]\!]$. Observe that a more precise state $P''$ can be obtained by intersecting each translated state separately with $[\![x \leq y]\!]$, that is, by calculating $(P_0 \sqcap [\![x \leq y]\!]) \sqcup ((P_{-1} \rhd x := x + 256) \sqcap [\![x \leq y]\!]) \sqcup ((P_1 \rhd x := x - 256) \sqcap [\![x \leq y]\!])$. This state, depicted as the grey area in the lower right graph, is smaller than $P'$ since $P_{-1}$ does not contribute at all. Indeed, this example shows that polyhedra are not meet-distributive, that is, $P \sqcap (P_1 \sqcup P_2) \neq (P \sqcap P_1) \sqcup (P \sqcap P_2)$. In this work, we chose to calculate the equivalent of $P'$ in our wrapping function *wrap* as it simplifies the presentation; implementing the refined model is mere engineering.

### 5.2 Wrapping Variables with Infinite Ranges

In the given example, it was possible to obtain a wrapped representation of the values of $x$ and $y$ by calculating the join of three constituent state spaces. In general, however, wrapping $x$ and $y$ can require the join of an infinite number of constituent state spaces as depicted in Figure 5. Here, the line in the upper graph depicts $P = [\![x + 1024 = 8y]\!]$, that is, $P$ denotes the same linear relation as before, except that $x$ is unbounded. Translating $P$ by $i$ times the range of **uint8** yields $P_i = (P \rhd x := x + i2^8 \sqcap [\![0 \leq x \leq 255]\!]) \sqcup (P \rhd x := x - i2^8 \sqcap [\![0 \leq x \leq 255]\!])$ for $i \geq 0$. A polyhedron that includes the sequence $P'_j = \bigsqcup_{0 \leq i \leq j} P_i$ can be computed using widening [6], thereby yielding the grey area in the lower right graph. In fact, this region is equivalent to $\exists_x(P) \sqcap [\![0 \leq x \leq 255]\!]$ as it contains neither bounds on $x$ nor relational information between $x$ and other variables. This suggests that, rather than wrapping unbounded variables, it is cheaper and

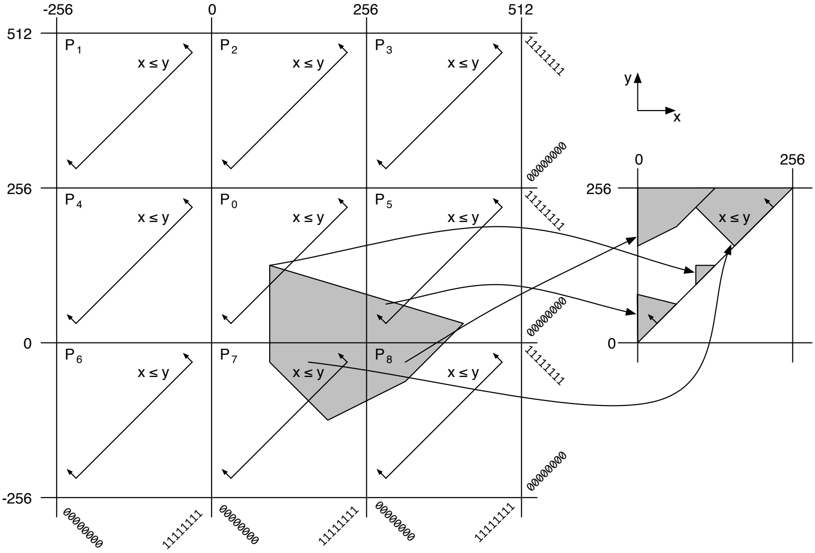


**Fig. 6.** Precise wrapping of two bounded variables

as precise to set them to the whole range of their type. After wrapping $x$, it becomes apparent that $y$ is unbounded too, and hence needs wrapping.

### 5.3 Wrapping Several Variables

Even though the guard `x<=y` used in the example of Section 5.1 involves two variables, it was only necessary to wrap $x$ to obtain a wrapped representation of both $x$ and $y$. The example of Section 5.2 hints at the fact that both variables might need wrapping to ensure that both sides of the guard are within range. In particular, it is not possible to translate a guard `x<=y` to the inequality $x - y \leq 0$ and to merely wrap $x - y$ to $[0, 255]$. To illustrate this, consider the simpler case `x<=42` which is satisfied for bit sequences of `x` that fall within $[0, 42]$. In order to evaluate `x<=42`, set $x' = x - 42$ and wrap $x'$ such that $0 \leq x' \leq 255$. The intersection with $[\![x' \leq 0]\!]$ constrains $x'$ to 0 which implies $x = 42$ instead of $x \in [0, 42]$. Thus, both arguments to a guard `x<=y` need to be wrapped independently.

The example in Figure 5 showed how wrapping the unbounded $x$ leaves $y$ unconstrained which thus has to be wrapped as well. Figure 6 shows a potentially more precise solution for bounded variables in which variables are wrapped simultaneously. Here, the bounded state space shown in grey expands beyond the state $P_0$ that corresponds to the actual range of the variables. The result of translating each neighbouring quadrant and intersecting it with $x \leq y$ is shown in the graph on the right. Note that the join of these four translated spaces retains no information on either $x$ or $y$. While it is possible that relational information with other variables is retained, wrapping the variables independently has the same precision if one of the variables is within bounds and, in particular,

**Algorithm 1.** Explicitly wrapping an expression to the range of a type

```
procedure $wrap(P, t\ s, x)$ where $P \neq \emptyset, t \in \{\text{uint}, \text{int}\}$ and $s \in \{1, 2, 4, 8\}$
 1: $b_l \leftarrow 0$
 2: $b_h \leftarrow 2^s$
 3: if $t =$ int then  /* Adjust ranges when wrapping to a signed type. */
 4:     $b_l \leftarrow b_l - 2^{s-1}$
 5:     $b_h \leftarrow b_h - 2^{s-1}$
 6: end if
 7: $[l, u] \leftarrow P(x)$
 8: if $l \neq -\infty \wedge u \neq \infty$ then  /* Calculate quadrant indices. */
 9:     $\langle q_l, q_u \rangle \leftarrow \langle \lfloor (l - b_l)/2^s \rfloor, \lfloor (u - b_l)/2^s \rfloor \rangle$
10: end if
11: if $l = -\infty \vee u = \infty \vee (q_u - q_l) > k$ then  /* Set to full range. */
12:     return $\exists_x(P) \sqcap [\![ b_l \leq x < b_h ]\!]$
13: else  /* Shift and join quadrants $\{q_l, \ldots q_u\}$. */
14:     return $\bigsqcup_{q \in [q_l, q_u]} ((P \rhd x := x - q2^s) \sqcap [\![ b_l \leq x < b_h ]\!])$
15: end if
```

if a variable is compared to a constant. In the 3000 LOC program `qmail-smtp` that our analysis targets, 427 out of 522 conditionals test a variable against a constant, which motivates our design choice of wrapping variables independently.

### 5.4 An Algorithm for Explicit Wrapping

Guided by the observations made in the three examples, Algorithm 1 gives a procedure to wrap a polyhedral variable to the range of a given integer type. Due to the observations in the last section, we only present an algorithm to wrap one variable at a time. Thus, both sides of a guard have to be wrapped individually.

The algorithm commences by calculating the bounds of the type $t\ s$. A **uint8** type, for instance, will set $b_l = 0$ and $b_h = 2^8 = 256$ while an **int8** type results in the bounds $b_l = 0 - 2^{8-1} = -128$ and $b_h = 2^8 - 2^{8-1} = 128$. Line 7 calculates the bounds of $x$ in $P$. If one of these bounds is infinite, line 12 removes all information on $x$ and restrains $x$ to $[b_l, b_h - 1]$. In case of finite bounds, line 9 calculates the smallest and largest quadrant into which the values of $x$ impinge. For instance, in the example of Figure 4, these numbers are $q_l = -1$ (for the quadrant $[-256, -1]$) and $q_h = 1$ (for $[256, 511]$). Line 11 ensures that the linear expression is simply set to its maximum bounds if more than $k$ quadrants have to be transposed and joined, where $k$ is a limit that can be tuned to the required precision. Line 14 transposes each quadrant and restricts it to the bounds of the type. The correctness of *wrap* is asserted below:

**Proposition 2.** *Given $P \neq \emptyset$ and $P' = wrap(P, t\ s, x_i)$, the interval $P'(x_i)$ lies in the range of the type $t\ s$. Furthermore $\gamma^{\boldsymbol{s}}_{\boldsymbol{a}}(P) \subseteq \gamma^{\boldsymbol{s}}_{\boldsymbol{a}}(P')$.*

*Proof.* Upon return from lines 12 and 14, $x_i$ is restricted to lie between the bounds $b_l$ and $b_h - 1$ of the type $t\ s$, hence $P'(x_i)$ lies in the range of type $t\ s$.

Suppose $\boldsymbol{a} = \langle a_1, \dots a_n \rangle$, $\boldsymbol{s} = \langle s_1, \dots s_n \rangle$ where $a_i = \mathit{addr}^\natural(x_i)$ and $s_i$ denotes the size of $x_i$ in bytes. Let $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P)$. Then there exists $\langle v_1, \dots v_n \rangle \in P \cap \mathbb{Z}^n$ such that $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(\{\boldsymbol{v}\})$. We consider two behaviours of *wrap*:

- Suppose that *wrap* is exited at line 12. Observe that for any $\boldsymbol{b} \in \mathbb{B}^{8s_i}$ there exists $v \in \{b_l, \dots, b_h - 1\}$ such that $\mathit{bin}^{8s_i}(v) = \boldsymbol{b}$. Hence, there exists $\boldsymbol{v'} = \langle v_1, \dots v'_i, \dots v_n \rangle$ with $v'_i \in [b_l, b_h - 1] \cap \mathbb{Z}$ and $\mathit{bin}^{8s_i}(v'_i) = \mathit{bin}^{8s_i}(v_i)$. Observe that $\boldsymbol{v'} \in P' = \exists_x(P) \sqcap [\![b_l \leq x < b_h]\!]$ and $\boldsymbol{v'} \in \mathbb{Z}^n$. Hence $\mathit{bits}_{\boldsymbol{a}}^{\boldsymbol{s}}(v_j) = \mathit{bits}_{\boldsymbol{a}}^{\boldsymbol{s}}(v'_j)$ for all $j \in [1, n]$ and it follows that $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P')$.
- Suppose now that *wrap* exits at line 14. Observe that $v_i \in [l, u]$ hence there exists $q \in [q_l, q_u]$ such that $v_i - q2^{s_1} \in [b_l, b_h - 1]$. Hence, there exists $\boldsymbol{v'} = \langle v_1, \dots v'_i, \dots v_n \rangle \in P'$ such that $v'_i = v_i - q2^{s_i}$. Since $\mathit{bin}^{8s_i}(q2^{s_i}) = \mathbf{0}$ it follows that $\mathit{bin}^{8s_i}(v'_i) = \mathit{bin}^{8s_i}(v_i - q2^{s_i}) = \mathit{bin}^{8s_i}(v_i)$. Thus $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P')$.

Note the translation of quadrants using $P \rhd x := x + q2^s$ can be implemented by a potentially cheaper affine transformation [2]. However, the shown solution can be readily implemented using other polyhedral domains [15,17] that do not directly support affine translations.

## 6 An Abstract Semantics for $\mathcal{L}$(ELang)

This section defines the abstract semantics of $\mathcal{L}$(ELang) in which a single polyhedron $P$ is calculated for each label $l$ where each label marks the beginning of a basic block. Starting with the unrestricted polyhedron $\mathbb{R}^{|X|}$ for the first basic block and with the empty polyhedron $\emptyset \subseteq \mathbb{R}^{|X|}$ for all others, the semantic function of the basic blocks are repeatedly evaluated until a (post-)fixpoint is reached [4]. We omit a formal definition of this fixpoint for simplicity. Once a fixpoint is reached, each state $\sigma$ that may arise in the concrete program at $l$ satisfies $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P)$ where $P$ is the polyhedron associated with the label $l$.

The first rule in Figure 7 specifies how the evaluation of statements feeds into the evaluation of control-flow statements. Specifically, $[\![\ \mathit{lookupNext}(l)\ ]\!]^\sharp_{\mathrm{Next}} P_l$ yields tuples such as $\langle P'_{l'}, l' \rangle$ indicating that $P'_{l'}$ must be joined with the existing state $P_{l'}$ at $l'$. For instance, **jump** $l$ merely returns the current state paired with the target label. The conditional calculates two new polyhedra $P^{then}$ (which is returned for the label $l$) and $P^{else}$ (which is used to evaluate other control-flow instructions). The calculation of $P^{else}$ makes use of a function *neg* which negates a relational operator, for example, $\mathit{neg}('<') = '\geq'$. The auxiliary function *cond* wraps the two arguments of the relational operator *op*. Like *wrap*, this function can only wrap single polyhedral variables which requires that *exp* is assigned to a temporary variable $y$ which is projected out once the guard is applied.

Observe that enforcing the guard by intersecting with $[\![x\ \mathit{op}\ y]\!]$ has the same effect as wrapping the expression *exp* itself since $y = \mathit{exp}$ holds in $P'$. However, if *wrap* returns from line 12 in Algorithm 1, the variable $y$ is merely set to the bounds of the type. In this case *wrap* discards the relational information between $y$ and *exp* and the intersection with $[\![x\ \mathit{op}\ y]\!]$ has no effect on $P''$, thereby ignoring the condition. An alternative treatment for expressions *exp* that exceed

$k$ quadrants would be to discard any previous information on variables in *exp* using projection and to modify *wrap* to intersect $P$ with $[\![b_l \leq exp < b_h]\!]$. In this case the information in the guard could be retained by intersecting with $[\![x\ op\ exp]\!]$ at the cost of discarding any previous bounds on the variables of *exp*.

The following Proposition states the correctness of the conditional:

**Proposition 3.** *If* $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P)$ *and* $val^{8s,t}(\sigma^s(addr^\natural(x_i)))\ op\ val^{8s,t}([\![\ exp\ ]\!]^{\natural,s}_{\text{Expr}}\sigma)$ *then* $\langle P', l\rangle \in [\![\ \textbf{if}\ t\ s\ x_i\ op\ exp\ \textbf{then jump}\ l\ ;\ nxt\ ]\!]^\sharp_{\text{Next}} P$ *and* $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P')$.

*Proof.* Since $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P)$ there exists $\boldsymbol{v} \in P \cap \mathbb{Z}^n$ such that $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(\{\boldsymbol{v}\})$. Let $\hat{P} = cond(\bar{P}, t\ s, x_i, y, op)$ where $\bar{P} = P \rhd y := [\![\ exp\ ]\!]^\sharp_{\text{Expr}}$. Then $\langle v_1, \ldots v_n, \hat{v}\rangle \in \bar{P} \cap \mathbb{Z}^{n+1}$ where $\langle v_1, \ldots v_n\rangle = \boldsymbol{v}$, $\hat{v} = \boldsymbol{c} \cdot \boldsymbol{v} + d$ and $exp \equiv \boldsymbol{c} \cdot \boldsymbol{x} + d$. By Proposition 2, there exists $\boldsymbol{v'} = \langle v_1, \ldots v_{i-1}, v'_i, v_{i+1}, \ldots v_n, \hat{v}'\rangle \in \hat{P} \cap \mathbb{Z}^{n+1}$ such that $bin^{8s}(v_i) = bin^{8s}(v'_i) = \sigma^{8s}(addr^\natural(x_i))$. By following Proposition 1, $bin^{8s}(\hat{v}) = bin^{8s}(\hat{v}') = [\![\ exp\ ]\!]^{\natural,s}_{\text{Expr}}\sigma$. Furthermore, $v'_i$ and $\hat{v}'$ lie in the range of $t\ s$ and thus $val^{8s,t}(\sigma^{8s}(addr^\natural(x_i))) = v'_i$ and $val^{8s,t}([\![\ exp\ ]\!]^{\natural,s}_{\text{Expr}}\sigma) = \hat{v}'$. Hence $\boldsymbol{v'} \in \hat{P} \sqcap [\![x\ op\ y]\!]$ for $op \notin \{\neq\}$. With $P' = \exists_y(\hat{P} \sqcap [\![x\,op\,y]\!])$ it follows that $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P')$. The argument is similar for $op \in \{\neq\}$.

The fall-through case can be shown correct by a similar argument.

Due to the modulo nature of $\gamma_{\boldsymbol{a}}^{\boldsymbol{s}}$ the evaluation of linear expressions and assignments resembles that of classic polyhedral analysis in that linear expressions in the program are simply re-interpreted as expressions over polyhedra variables. This holds true even for casts between different sized variables as long as the target variable is smaller. Assigning smaller variables to larger, on the contrary, requires that wrapping is made explicit since a value that exceeds the range of the smaller source variable would wrap in the actual program whereas it might not exceed the range of the larger target variable.

We conclude this section with a correctness argument for the cast statement:

**Proposition 4.** *Suppose* $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P)$, $\sigma' = [\![\ s_1\ x_i = t\ s_2\ x_j\ ]\!]^\natural_{\text{Stmt}}\sigma$ *and let* $P' = [\![\ s_1\ x_i = t\ s_2\ x_j\ ]\!]^\sharp_{\text{Stmt}} P$. *Then* $\sigma' \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P')$.

*Proof.* Since $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P)$ there exists $\boldsymbol{v} = \langle v_1, \ldots v_n\rangle \in P$ such that $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(\{\boldsymbol{v}\})$. Let $\langle v'_1, \ldots v'_n\rangle \in P \rhd x_i := x_j$ where $v'_i = v'_j$ and $v'_k = v_k$ for all $k \neq i$. By Lemma 1, $\sigma'^{s_k}(addr^\natural(x_k)) = \sigma^{s_k}(addr^\natural(x_k))$ for all $k \neq i$. By definition of $[\![\ \cdot\ ]\!]^\natural_{\text{Stmt}}$, we need to show that $\sigma'^{s_1}(addr^\natural(x_i)) = bin^{8s_1}(val^{8s_2,t}(\sigma^{s_2}(addr^\natural(x_j))))$.

- Suppose $s_1 \leq s_2$. Then $bin^{8s_1}(x) = bin^{8s_1}(val^{8s_2,t}(bin^{8s_2}(x)))$. But $bin^{8s_1}(v'_i) = \sigma'^{s_1}(addr^\natural(x_i))$ and $bin^{8s_2}(v_j) = \sigma^{s_2}(addr^\natural(x_j))$, thus $\sigma' \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P')$ follows.
- Suppose now that $s_1 > s_2$. By Proposition 2, there exists $\langle v_1, \ldots \hat{v}_i, \ldots v_n\rangle \in P'$ such that $bin^{8s_2}(\hat{v}_i) = bin^{8s_2}(v_j)$ and $\hat{v}_i$ lies in the range of $t\ s_2$, that is, $val^{8s_2,t}(bin^{8s_2}(\hat{v}_i)) = \hat{v}_i$. But since $bin^{8s_2}(\hat{v}_i) = bin^{8s_2}(v_j) = \sigma'^{s_1}(addr^\natural(x_j))$ it follows that $\sigma'^{s_1}(addr^\natural(x_i)) = bin^{8s_1}(\hat{v}_i)$ as required.

Basic Blocks.

$$[\![\ l : s_1; \ldots s_n;\ ]\!]^\sharp_{\text{Block}} P = [\![\ \mathit{lookupNext}(l)\ ]\!]^\sharp_{\text{Next}}([\![\ s_n\ ]\!]^\sharp_{\text{Stmt}}(\ldots [\![\ s_1\ ]\!]^\sharp_{\text{Stmt}} P \ldots))$$

Control Flow.

$$[\![\ \textbf{jump}\ l\ ]\!]^\sharp_{\text{Next}} P = \{\langle P, l\rangle\}$$

$$[\![\ \textbf{if}\ t\ s\ v\ \mathit{op}\ \mathit{exp}\ \textbf{then jump}\ l\ ;\ \mathit{nxt}\ ]\!]^\sharp_{\text{Next}} P = \{\langle P^{then}, l\rangle\} \cup [\![\ \mathit{nxt}\ ]\!]^\sharp_{\text{Next}} P^{else}$$

$$\begin{aligned} \text{where } P' &= P \rhd y := [\![\ \mathit{exp}\ ]\!]^\sharp_{\text{Expr}} \text{ where } y \in T \text{ fresh} \\ P^{then} &= \exists_y(\mathit{cond}(P', t\ s, v, y, \mathit{op})) \\ P^{else} &= \exists_y(\mathit{cond}(P', t\ s, v, y, \mathit{neg}(\mathit{op}))) \end{aligned}$$

$$\mathit{cond}(P', t\ s, x, y, \mathit{op}) = \begin{cases} (P'' \sqcap [\![x < y]\!]) \sqcup (P'' \sqcap [\![x > y]\!]) & \text{if } \mathit{op} \in \{\neq\} \\ (P'' \sqcap [\![x\ \mathit{op}\ y]\!]) & \text{otherwise} \end{cases}$$

$$\text{where } P'' = \mathit{wrap}(\mathit{wrap}(P', t\ s, x), t\ s, y)$$

Expressions.

$$[\![\ n\ ]\!]^\sharp_{\text{Expr}} = n$$

$$[\![\ n * v + \mathit{exp}\ ]\!]^\sharp_{\text{Expr}} = n\ v + [\![\ \mathit{exp}\ ]\!]^\sharp_{\text{Expr}}$$

Assignments.

$$[\![\ s\ v = \mathit{exp}\ ]\!]^\sharp_{\text{Stmt}} P = P \rhd v := [\![\ \mathit{exp}\ ]\!]^\sharp_{\text{Expr}}$$

Type Casts.

$$[\![\ s_1\ v_1 = t\ s_2\ v_2\ ]\!]^\sharp_{\text{Stmt}} P = \begin{cases} P' & \text{if } s_1 \leq s_2 \\ \mathit{wrap}(P', t\ s_2, v_1) & \text{otherwise} \end{cases}$$

$$\text{where } P' = P \rhd v_1 := v_2$$

**Fig. 7.** Abstract semantics of $\mathcal{L}$(ELang)

## 7 Discussion

The existence of a concretisation map $\gamma^{\boldsymbol{s}}_{\boldsymbol{a}}$ begs the question of whether an abstraction map can be defined. For classic polyhedral analysis [6], it is well-known that no best abstraction exists for certain shapes such as a disc [4]. In the context of our analysis, the set of concrete states $\Sigma$ is finite. However, a given set of states still has no best abstraction. Consider $\sigma \in \Sigma$ with $\sigma^1(\mathit{addr}^\natural(\mathtt{x})) = 11111111$, $P_1 = [\![x = -1]\!]$ and $P_2 = [\![x = 255]\!]$. Although $\sigma \in \gamma^{\boldsymbol{s}}_{\boldsymbol{a}}(P_1) = \gamma^{\boldsymbol{s}}_{\boldsymbol{a}}(P_2)$, $P_1$ and $P_2$ are incomparable. As a consequence, the meet operation can only be applied after *wrap* has expressed the polyhedra in the same quadrant and thereby made them comparable. Termination is not compromised as *wrap* is monotonic.

Since different polyhedra can describe the same set of concrete states, care is needed when applying join. Suppose that the loop in Figure 8 is entered with $P = [\![x = -1]\!]$: the largest value an unsigned variable can take. As the loop invariant $x \geq 42$ mentions $x$, $Q = P \sqcup U$ is wrapped to $R = \mathit{wrap}(Q, \mathbf{uint8}, x) = [\![x = 255]\!]$. A precision loss occurs when $P$ and $U = [\![x = 254]\!]$ are joined to obtain $[\![-1 \leq x \leq 254]\!]$ as $x$ cannot fall below 42. One solution to this particular problem is to unroll the loop once, which avoids the join of different representatives.

Observe that *wrap* is idempotent and, as such, is the identity if the variable is in range. An important consequence is that our solution is as precise as classic polyhedral analysis if all variables remain within the range of their types.

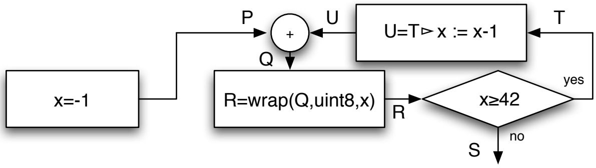


**Fig. 8.** Precision loss incurred by joining flow paths

An interesting benefit of $\gamma^s_a$ is that the possible values of a byte $x$ can be represented as either $[\![-128 \leq x \leq 127]\!]$ or $[\![0 \leq x \leq 255]\!]$. For example, an analysis of C string buffers [14] does not model individual array elements but tracks a single NUL position (a character with value zero) within the array. Even though `char` is often signed, the range $[0, 255]$ (rather than $[-128, 127]$) can be returned to indicate an arbitrary value when reading a byte from the array. The unsigned range can then be refined using the NUL position to $[1, 255]$ whenever the access lies in front of the NUL position (see [16] for an example). If a signed range $[-128, 127]$ had to be returned, it would include the NUL character since $[-128, -1] \cup [1, 127] = [-128, 127]$ is the best convex approximation. Without this tactic, it can be difficult to prove termination of loops that iterate over strings.

## 8 Related Work and Conclusion

A sound analysis must reason about overflowing calculations and correctly model conversions between unsigned and signed integers. Rather than contaminating an analysis with the low-level details of two's complement wrapping behaviour, we presented a computationally light-weight solution based on a novel concretization map for polyhedra which eliminates the need to check for wrapping in assignments of linear expression and conversions to smaller integers. Conditionals and other casts require an implementation using the presented algorithm *wrap*. We proved the presented analysis correct and argued the precision is no worse than that of classic polyhedral analysis that warns about every wrapping.

Although a number of works have addressed the modulo properties of congruences [1,11,12,13], little work exists for polyhedral analyses. Blanchet *et al.* use a two-tier approach [5]: For signed integers, any wrapping is erroneous. In this case, each time a variable is set, its range is checked for overflows. Overflows of unsigned integers are assumed to be intentional as wrapping may result from bit-level operations. This approach requires a separation of signed and unsigned variables which incurs false warnings for many programs, such as the C program in our introduction. Indeed, an analysis of our example would lead to the misleading warning about converting from a signed to an unsigned integer.

Further afield is work on tracing the propagation of rounding errors in floating point calculations. In this context, Goubault et al. [10] treat wrapping of integers as a rounding error that is as large as the range of the integer variable.

*Acknowledgements.* This work is supported by EPSRC project EP/C015517.

## References

1. Bagnara, R., Dobson, K., Hill, P.M., Mundell, M., Zaffanella, E.: Grids: A Domain for Analyzing the Distribution of Numerical values. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 219–235. Springer, Heidelberg (2007)
2. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.M.: Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 213–229. Springer, Heidelberg (2002)
3. Joint Technical Committee.: International Standard ISO/IEC of C 98/99 (1999)
4. Cousot, P., Cousot, R.: Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
5. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE Analyzer. In: European Symposium on Programming, pp. 21–30. Springer, Edinburgh, Scotland (2005)
6. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Constraints among Variables of a Program. In: Principles of Programming Languages, pp. 84–97. ACM Press, Tucson, Arizona (1978)
7. Davenport, H.: The Higher Arithmetic, 7th edn. Cambridge University Press, Cambridge (1952)
8. Dor, N., Rodeh, M., Sagiv, M.: Cleanness Checking of String Manipulations in C Programs via Integer Analysis. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 194–212. Springer, Heidelberg (2001)
9. Dor, N., Rodeh, M., Sagiv, M.: CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. In: Gupta, R. (ed.) Programming Language Design and Implementation, pp. 155–167. ACM Press, San Diego, USA (2003)
10. Goubault, E., Putot, S., Beaufreton, P., Gassino, J.: Static Analysis of the Accuracy in Control Systems: Principles and Experiments. In: FMICS 2007. 12th International Workshop on Formal Methods for Industrial Critical systems, Springer, Heidelberg (2007)
11. Granger, P.: Static Analysis of Arithmetic Congruences. International Journal of Computer Mathematics 30, 165–199 (1989)
12. Granger, P.: Static Analyses of Congruence Properties on Rational Numbers (Extended Abstract). In: Symposium on Static Analysis, pp. 278–292. Springer, London, UK (1997)
13. Müller-Olm, M., Seidl, H.: Analysis of Modular Arithmetic. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 46–60. Springer, Heidelberg (2005)
14. Simon, A., King, A.: Analyzing String Buffers in C. In: Kirchner, H., Ringeissen, C. (eds.) AMAST 2002. LNCS, vol. 2422, pp. 365–379. Springer, Heidelberg (2002)
15. Simon, A., King, A.: Exploiting Sparsity in Polyhedral Analysis. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 336–351. Springer, Heidelberg (2005)
16. Simon, A., King, A.: Widening Polyhedra with Landmarks. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 166–182. Springer, Heidelberg (2006)
17. Simon, A., King, A., Howe, J.M.: Two Variables per Linear Inequality as an Abstract Domain. In: Leuschel, M.A. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 71–89. Springer, Heidelberg (2003)

# Under-Approximations of Computations in Real Numbers Based on Generalized Affine Arithmetic

Eric Goubault and Sylvie Putot

CEA, LIST
Laboratory for ModEling and Analysis of Systems in Interaction,
Boîte courrier 65, GIF-SUR-YVETTE CEDEX, F-91191 France
{eric.goubault,sylvie.putot}@cea.fr

**Abstract.** We build a new, implicitly relational abstract domain which gives accurate under-approximations of the set of real values that program variables can take. This statement is demonstrated both on a theoretical basis and on non-trivial numerical examples. It is, we believe, the first non-trivial under-approximating numerical domain in the static analysis literature.

## 1 Introduction

Most abstract interpretation *numerical domains* construct over-approximations of the range of program variables. This is the case for intervals [3], zones [13], polyhedra [5] etc. Of course, such static analyses are essentially Galois connection based, formalism which precisely expresses the over-approximation process. One needs dual Galois connections, or dual concretization based frameworks, as developed in e.g. [16], to express under-approximations.

In this paper, we develop an abstract interpretation domain for directly under-approximating the range of real values of program variables. It is based on a variant of the affine form domain developed by the authors for over-approximations [9], and on ideas from generalized interval arithmetic [7,8].

Such under-approximations, when combined with over-approximations, give an estimate of the quality of the result of a static analysis. But of course, our work can also be applied to statically find run-time errors that are bound to occur, from some given set of possible initial states. It can also be applied to the analysis of temporal properties of reactive systems. The latter point was studied in [6], and formalized through Galois and dual Galois connections in [15]. It is also studied in abstract model-checking, see for instance [10,14]. We believe that these analyses can also benefit from our approach.

*Contents.* In section 2.1, we recall the main definitions and properties of generalized interval arithmetic, and its potential interpretations as under-approximations. We then extend these ideas to affine forms. In section 2.3, we use a generalized mean-value theorem [8] to define an under-approximating semantics

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 137–152, 2007.

of arithmetic expressions. This semantics, though applied to forms that are very close to the affine forms used for over-approximation in [9], is very different from the semantics proposed in [9], and yields a direct under-approximation of the result of arithmetic expressions. We develop the order-theoretic apparatus needed for an actual static analysis in section 2.4, and apply this analysis in section 3.

*Main contributions.* We describe a new numerical abstract domain that gives accurate under-approximations of the values of program variables. The time and space complexities of the primitive operations are small, as was the case with the approach for over-approximations of [9], which bear interesting relationships with the present work. Indeed, the interest of combining the two analyses is exemplified. Using the prototype we implemented, we demonstrate very good precision of the analysis on non trivial numerical programs. On linear recursive filters of any order, pervasive in all control programs, we demonstrate how the analysis results for the output variable can be made as close as we want to the real range, see lemma 1. We also demonstrate in the case of linear recursive filters, how the abstract invariant discovered by our method allows us to find a sequence of inputs over time that lead to a value as close as we want to the maximal or minimal output value, allowing us to produce witnesses of potentially bad behaviors. An even more general result holds for arbitrary reactive programs, see lemma 2, and is exemplified on a perturbed filter.

## 2 An Under-Approximating Domain Based on Generalized Affine Arithmetic

### 2.1 Generalized Affine Forms

We first introduce the principles of generalized interval arithmetic, following [7,8], and their interpretation using quantifiers. We refer the reader to these recent papers, that revisit the ideas of modal intervals, for more references on generalized intervals, modal intervals and Kaucher arithmetic [11,12].

Our contribution here is to then extend these ideas to generalize affine forms, and interpret them either as over or under-approximating forms for real values of variables.

**Generalized interval arithmetic and notations.** Generalized intervals are intervals whose bounds are not ordered. The set of classical intervals is denoted by $\mathbb{IR} = \{[a, b],\ a \in \mathbb{R}, b \in \mathbb{R}, a \leq b\}$. The set of generalized intervals is denoted by $\mathbb{IK} = \{[a, b],\ a \in \mathbb{R}, b \in \mathbb{R}\}$. Intervals (classical or generalized) will be noted with bold letters.

Related to a set of real numbers $\{x \in \mathbb{R},\ a \leq x \leq b\}$, one can consider two generalized intervals, $[a, b]$, which is called proper, and $[b, a]$, which is called improper. We define the operations dual $[a, b] = [b, a]$ and pro $[a, b] = [\min(a, b), \max(a, b)]$.

The generalized intervals are partially ordered by inclusion which extends inclusion of classical intervals. Intervals (classical or generalized) will be noted with bold letters. Given two generalized intervals $\boldsymbol{x} = [\underline{x}, \overline{x}]$ and $\boldsymbol{y} = [\underline{y}, \overline{y}]$, the inclusion is defined by

$$\boldsymbol{x} \sqsubseteq \boldsymbol{y} \Leftrightarrow \underline{y} \leq \underline{x} \wedge \overline{x} \leq \overline{y}.$$

The inclusion is then related to the dual operation by $\boldsymbol{x} \sqsubseteq \boldsymbol{y} \Leftrightarrow \text{dual } \boldsymbol{x} \sqsupseteq \text{dual } \boldsymbol{y}$. Kaucher addition extends addition on classical intervals:

$$\begin{aligned}
&\boldsymbol{x} + \boldsymbol{y} = [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \\
&\boldsymbol{x} - \boldsymbol{y} = [\underline{x} - \overline{y}, \overline{x} - \underline{y}] = \boldsymbol{x} + (-\boldsymbol{y}) \text{ where } -\boldsymbol{y} = [-\overline{y}, -\underline{y}].
\end{aligned}$$

We let $\mathcal{P} = \{\boldsymbol{x} = [\underline{x}, \overline{x}],\ \underline{x} \geq 0 \wedge \overline{x} \geq 0\}$, $-\mathcal{P} = \{\boldsymbol{x} = [\underline{x}, \overline{x}],\ \underline{x} \leq 0 \wedge \overline{x} \leq 0\}$, $\mathcal{Z} = \{\boldsymbol{x} = [\underline{x}, \overline{x}],\ \underline{x} \leq 0 \leq \overline{x}\}$, and dual $\mathcal{Z} = \{\boldsymbol{x} = [\underline{x}, \overline{x}],\ \underline{x} \geq 0 \geq \overline{x}\}$. Kaucher multiplication $\boldsymbol{x} \times \boldsymbol{y}$ is described in table 1. Kaucher division is defined for all $y$ such that $0 \notin \text{pro } \boldsymbol{y}$ by $\boldsymbol{x}/\boldsymbol{y} = \boldsymbol{x} \times [1/\overline{y}, 1/\underline{y}]$. When restricted to proper intervals, these operations coincide with the classical interval operations. Kaucher arithmetic has better algebraic properties than classical interval arithmetic: Kaucher addition turns $\mathbb{IK}$ into a group, as $\boldsymbol{x} + (-\text{dual } \boldsymbol{x}) = 0$. Kaucher multiplication turns $\mathbb{IK}$ restricted to generalized intervals whose products of bounds are strictly positive into a group, as $\boldsymbol{x} \times (1/\text{dual } \boldsymbol{x}) = 1$.

**Table 1.** Kaucher multiplication ([11,12])

| $\boldsymbol{x} \times y$ | $\boldsymbol{y} \in \mathcal{P}$ | $\boldsymbol{y} \in \mathcal{Z}$ | $\boldsymbol{y} \in -\mathcal{P}$ | $y \in \text{dual}\mathcal{Z}$ |
|---|---|---|---|---|
| $\boldsymbol{x} \in \mathcal{P}$ | $[\underline{x}\underline{y}, \overline{x}\overline{y}]$ | $[\overline{x}\underline{y}, \overline{x}\overline{y}]$ | $[\overline{x}\underline{y}, \underline{x}\overline{y}]$ | $[\underline{x}\underline{y}, \underline{x}\overline{y}]$ |
| $\boldsymbol{x} \in \mathcal{Z}$ | $[\underline{x}\overline{y}, \overline{x}\overline{y}]$ | $[\min(\underline{x}\overline{y}, \overline{x}\underline{y}),$<br>$\max(\underline{x}\underline{y}, \overline{x}\overline{y})]$ | $[\overline{x}\underline{y}, \underline{x}\underline{y}]$ | $0$ |
| $\boldsymbol{x} \in -\mathcal{P}$ | $[\underline{x}\overline{y}, \overline{x}\underline{y}]$ | $[\underline{x}\overline{y}, \underline{x}\underline{y}]$ | $[\overline{x}\overline{y}, \underline{x}\underline{y}]$ | $[\overline{x}\overline{y}, \overline{x}\underline{y}]$ |
| $\boldsymbol{x} \in \text{dual}\mathcal{Z}$ | $[\underline{x}\underline{y}, \overline{x}\underline{y}]$ | $0$ | $[\overline{x}\overline{y}, \underline{x}\overline{y}]$ | $[\max(\underline{x}\underline{y}, \overline{x}\overline{y}),$<br>$\min(\underline{x}\overline{y}, \overline{x}\underline{y})]$ |

**Interpretation of interval computations using quantifiers (see [8]).** Classical interval computations can be interpreted as quantified propositions. As an example, take $f$ to be the function defined by $f(x) = x^2 - x$. Extended to interval arithmetic, its value on $x = [2, 3]$ is $f([2, 3]) = [2, 3]^2 - [2, 3] = [1, 7]$, which can be interpreted as the proposition

$$(\forall x \in [2, 3])\,(\exists z \in [1, 7])\,(f(x) = z).$$

Modal intervals extend classical intervals by coupling a quantifier to them. Extensions of modal intervals were proposed (see [7]) in the framework of generalized intervals, and called AE extensions because universal quantifiers (All) always precede existential ones (Exist) in the interpretations. They give rise to a generalized interval arithmetic which coincides with Kaucher arithmetic. Let $f : \mathbb{R}^n \to \mathbb{R}$ a function in which each variable appears only once. Let $\boldsymbol{x} \in \mathbb{IK}^n$, which we can decompose in $\boldsymbol{x}_{\mathcal{A}} \in \mathbb{IR}^p$ and $\boldsymbol{x}_{\mathcal{E}} \in (\text{dual } \mathbb{IR})^q$ with $p + q = n$. We consider the problem of computing a quantifier $Q_z$ and an interval $\boldsymbol{z} \in \mathbb{IK}$ such that

$$(\forall x_{\mathcal{A}} \in \boldsymbol{x}_{\mathcal{A}})\,(Q_z z \in \text{pro } \boldsymbol{z})\,(\exists x_{\mathcal{E}} \in \text{pro } \boldsymbol{x}_{\mathcal{E}})(f(x) = z). \tag{1}$$

In these expressions, if $\boldsymbol{z}$ is proper then $Q_z = \exists$, else $Q_z = \forall$. When all intervals are proper, we retrieve the interpretation of classical interval computation, which gives an over-approximation of the range of $f(x)$:

$$(\forall x \in \boldsymbol{x})\,(\exists z \in \boldsymbol{z})\,(f(x) = z).$$

And when all intervals are improper, we get an under-approximation:

$$(\forall z \in \text{pro } \boldsymbol{z})\,(\exists x \in \text{pro } \boldsymbol{x})\,(f(x) = z).$$

**Affine forms for over and under-approximation.** An affine form ([17]) is a polynomial of degree one in a set of symbols $\varepsilon_i$ called noise symbols:

$$\hat{x} = \alpha_0^x + \alpha_1^x \varepsilon_1 + \ldots + \alpha_n^x \varepsilon_n, \;\text{ with } \alpha_i^x \in \mathbb{R}. \tag{2}$$

Each noise symbol $\varepsilon_i$ is a formal variable representing an independent component of the total uncertainty on the quantity $x$, its value unknown but bounded in $[-1, 1]$; the corresponding coefficient $\alpha_i^x$, called partial deviation, is a known real value. Coefficient $\alpha_0^x$ is the center of the affine form. The idea is that the same noise symbol can be shared by several quantities, expressing correlations between them.

In [9], we defined a domain for *over-approximation* of real values based on these forms with real coefficients. The concretization $\hat{\Gamma}(\hat{x})$ of $\hat{x}$ is a proper interval obtained by the evaluation of expression (2) with proper intervals $\boldsymbol{\varepsilon_i} = [-1, 1]$ and classical interval arithmetic:

$$\hat{\Gamma}(\hat{x}) = \alpha_0^x + \alpha_1^x \boldsymbol{\varepsilon_1} + \ldots + \alpha_n^x \boldsymbol{\varepsilon_n}.$$

We define here a domain for *under-approximation* based on generalized affine forms, where the $\boldsymbol{\alpha_i^x}$ coefficients are no longer real numbers but proper intervals:

$$\check{x} = \boldsymbol{\alpha_0^x} + \boldsymbol{\alpha_1^x} \varepsilon_1 + \ldots + \boldsymbol{\alpha_n^x} \varepsilon_n, \;\text{ with } \boldsymbol{\alpha_i^x} \in \mathbb{IR}. \tag{3}$$

We define the concretization $\check{\Gamma}(\check{x})$ of $\check{x}$ obtained by the evaluation of expression (3) with improper intervals $\boldsymbol{\varepsilon_i^*} = [1, -1]$ and Kaucher interval arithmetic:

$$\check{\Gamma}(\check{x}) = \boldsymbol{\alpha_0^x} + \boldsymbol{\alpha_1^x} \boldsymbol{\varepsilon_1^*} + \ldots + \boldsymbol{\alpha_n^x} \boldsymbol{\varepsilon_n^*}.$$

We will construct semantics of arithmetic operations on these forms such that if $\check{\Gamma}(\check{x})$ is an improper interval, then it gives an under-approximation of the range of the real values taken by $x$. Otherwise, it cannot be interpreted as an under-approximation. If $\boldsymbol{\alpha_0^x}$ is an interval with zero width (i.e. a real number), then $\check{\Gamma}(\check{x})$ is always an improper interval (strictly improper or with zero width). Note that the extension $\hat{\Gamma}(\check{x})$ of $\hat{\Gamma}$ on $\check{x}$ will give an over-approximation of the values of $x$, but most of the time less precise than $\hat{\Gamma}(\hat{x})$ (see example 1 of section 2.3).

### 2.2 Semantics of Affine Operations

The result of linear operations on (generalized) affine forms can be exactly interpreted as an affine form, without additional under or over-approximation. For two variables $x$ and $y$ defined by affine forms (3), and a real number $r$, we get:

$$\begin{aligned}
x + y &= (\boldsymbol{\alpha_0^x} + \boldsymbol{\alpha_0^y}) + (\boldsymbol{\alpha_1^x} + \boldsymbol{\alpha_1^y})\varepsilon_1 + \ldots + (\boldsymbol{\alpha_n^x} + \boldsymbol{\alpha_n^y})\varepsilon_n \\
x + r &= (\boldsymbol{\alpha_0^x} + r) + \boldsymbol{\alpha_1^x}\varepsilon_1 + \ldots + \boldsymbol{\alpha_n^x}\varepsilon_n \\
r.x &= r\boldsymbol{\alpha_0^x} + r\boldsymbol{\alpha_1^x}\varepsilon_1 + \ldots + r\boldsymbol{\alpha_n^x}\varepsilon_n
\end{aligned}$$

Thus, if for example the ranges of $x$ and $y$ are known exactly, i.e. $\hat{\Gamma}(x) = \text{pro}\ \check{\Gamma}(x)$ and $\hat{\Gamma}(y) = \text{pro}\ \check{\Gamma}(y)$, then we also have $\hat{\Gamma}(x+y) = \text{pro}\ \check{\Gamma}(x+y)$, i.e. the range of the real result $x+y$ is known exactly (under the assumption that we compute in real numbers, see 3.1 for implementation details).

### 2.3 Semantics of Non Affine Arithmetic Operations

We use for the under-approximation of the result of non affine arithmetic operations, an extension of the mean-value theorem to generalized intervals (see [7,8]), which we extend to our generalized affine forms. We then derive two possible semantics for the under-approximation of the multiplication. Note that we could also, in the same way, derive semantics for other arithmetic operations.

**Mean-value theorem for generalized affine forms.** Suppose we have an affine model of variables $x_1, \ldots, x_k$, described as affine combinations such as (3) of noise symbols $\varepsilon_1, \ldots, \varepsilon_n$. For a differentiable function $f : \mathbb{R}^k \rightarrow \mathbb{R}$, we write $f^\varepsilon : \mathbb{R}^n \rightarrow \mathbb{R}$ the function induced by $f$ on $\varepsilon_1$ to $\varepsilon_n$. Suppose we have an over-approximation $\boldsymbol{\Delta_i}$ of the partial derivatives

$$\left\{ \frac{\partial f^\varepsilon}{\partial \varepsilon_i}(\varepsilon),\ \varepsilon \in [-1,1]^n \right\} \subseteq \boldsymbol{\Delta_i}. \tag{4}$$

Then

$$\tilde{f}^\varepsilon(\varepsilon_1, \ldots, \varepsilon_n) = f^\varepsilon(t_1, \ldots, t_n) + \sum_{i=1}^{n} \boldsymbol{\Delta_i}(\varepsilon_i - t_i), \tag{5}$$

where $(t_1, \ldots, t_n)$ is any point in $[-1,1]^n$, is interpretable in particular in the following sense :

- if $\tilde{f}^\varepsilon(\boldsymbol{\varepsilon_1^*}, \ldots, \boldsymbol{\varepsilon_n^*})$, computed with Kaucher arithmetic, is an improper interval, then pro $\tilde{f}^\varepsilon(\boldsymbol{\varepsilon_1^*}, \ldots, \boldsymbol{\varepsilon_n^*})$ is an under-approximation of $f^\varepsilon(\boldsymbol{\varepsilon_1}, \ldots, \boldsymbol{\varepsilon_n})$.
- if $\tilde{f}^\varepsilon(\boldsymbol{\varepsilon_1}, \ldots, \boldsymbol{\varepsilon_n})$ is a proper interval, then it is an over-approximation of $f^\varepsilon(\boldsymbol{\varepsilon_1}, \ldots, \boldsymbol{\varepsilon_n})$.

Note that a tighter estimation of $\boldsymbol{\Delta_i}$ can also be used (see [7]):

$$\left\{ \frac{\partial f^\varepsilon}{\partial \varepsilon_i}(\varepsilon_1, \ldots, \varepsilon_i, t_{i+1}, \ldots, t_n),\ (\varepsilon_1, \ldots, \varepsilon_i) \in [-1,1]^i \right\} \subseteq \boldsymbol{\Delta_i}. \tag{6}$$

Also, this theorem can be of course used when we take the $\varepsilon_i$ in sub-ranges of $[-1,1]$, it will in fact be used in examples to improve the accuracy of the results.

**Application to the multiplication.** We derive two affine under-approximating models for the multiplication. Model 1 is obtained using the Mean-Value Theorem on the real function $f^{\varepsilon}$ defined by the multiplication of the two real variables $x$ and $y$, which can be defined as real functions of the $\varepsilon_i$. Model 2 is obtained using it on the approximate model $g^{\varepsilon}$, in which the approximation is due to previous under-approximation of variables $x$ and $y$. As both models have advantages and drawbacks, we use a combination.

1. *Model 1, using (5) on the real function, with estimation (4) for $\boldsymbol{\Delta_i}$.* We can easily prove by recurrence that, for all variables $z$ whose real value is a linear function of noise symbols $\varepsilon_1, \ldots, \varepsilon_n$, the coefficient $\boldsymbol{\alpha_i^z}$ of the affine form obtained from our semantics is an over-approximation of $\frac{\partial z}{\partial \varepsilon_i}$. Then, for $f(x, y) = xy$, we can over-approximate
$$\frac{\partial f^{\varepsilon}}{\partial \varepsilon_i}(x, y) = \frac{\partial x}{\partial \varepsilon_i} y + \frac{\partial y}{\partial \varepsilon_i} x$$
by
$$\boldsymbol{\Delta_i} = \boldsymbol{\alpha_i^x y} + \boldsymbol{\alpha_i^y x}, \tag{7}$$
for any over-approximation $\boldsymbol{x}$ and $\boldsymbol{y}$ of the values taken by $x$ and $y$. However, the real value $f^{\varepsilon}(t_1, \ldots, t_n)$ has to be computed inductively, forbidding in practice a dynamic choice of the $t_i$. But the advantage is that it is computable exactly for any values chosen *a priori* of the $t_i$. Under the assumption that we compute in real numbers, the center $\alpha_0^{\check{z}}$ of the generalized affine forms used with this model is a real coefficient and not an interval. The concretization $\check{\Gamma}(\check{z})$ is thus always interpretable as an under-approximation.
2. *Model 2, using (5) on the approximate function, with improved estimation (4) for $\boldsymbol{\Delta_i}$.* We consider affine forms $\check{x}$ and $\check{y}$ giving an under-approximation when computed with improper $\varepsilon_i^*$. The approximate function $g^{\varepsilon}$ is given by
$$g^{\varepsilon}(\varepsilon) = \check{x} \times \check{y} = (\boldsymbol{\alpha_0^x} + \boldsymbol{\alpha_1^x}\varepsilon_1 \ldots + \boldsymbol{\alpha_n^x}\varepsilon_n)(\boldsymbol{\alpha_0^y} + \boldsymbol{\alpha_1^y}\varepsilon_1 \ldots + \boldsymbol{\alpha_n^y}\varepsilon_n).$$
This allows us to dynamically choose $t_1, \ldots, t_n$, as the evaluation $g^{\varepsilon}(t_1, \ldots, t_n)$ for any point $(t_1, \ldots, t_n)$ is straightforward. The affine form for the result of the multiplication $z = x \times y$ is then
$$\check{z} = (\boldsymbol{\alpha_0^x} + \boldsymbol{\alpha_1^x}t_1 \ldots + \boldsymbol{\alpha_n^x}t_n)(\boldsymbol{\alpha_0^y} + \boldsymbol{\alpha_1^y}t_1 \ldots + \boldsymbol{\alpha_n^y}t_n) + \sum_{i=1}^{n} \boldsymbol{\Delta_i}(\varepsilon_i - t_i). \tag{8}$$
In the general case, the center $g^{\varepsilon}(t_1, \ldots, t_n) = (\boldsymbol{\alpha_0^x} + \boldsymbol{\alpha_1^x}t_1 \ldots + \boldsymbol{\alpha_n^x}t_n)(\boldsymbol{\alpha_0^y} + \boldsymbol{\alpha_1^y}t_1 \ldots + \boldsymbol{\alpha_n^y}t_n)$ of this form is a proper interval, which may lead to a $\check{z}$ which is not interpretable as an under-approximation.

   Let us now compute the $\boldsymbol{\Delta_i}$. The partial derivatives over $\varepsilon_i$ of this product for any given real values $(a_0^x, a_1^x, \ldots, a_n^x, a_0^y, a_1^y, \ldots, a_n^y) \in (\boldsymbol{\alpha_0^x}, \boldsymbol{\alpha_1^x}, \ldots, \boldsymbol{\alpha_n^x}, \boldsymbol{\alpha_0^y}, \boldsymbol{\alpha_1^y}, \ldots, \boldsymbol{\alpha_n^y})$, is:

$$\frac{\partial g^\varepsilon}{\partial \varepsilon_i}(\varepsilon_1,\ldots,\varepsilon_i,t_{i+1},\ldots,t_n) = (a_i^x a_0^y + a_i^y a_0^x) + \sum_{j=1}^{i}(a_i^x a_j^y + a_i^y a_j^x)\varepsilon_j + \sum_{j=i+1}^{n}(a_i^x a_j^y + a_i^y a_j^x)t_j.$$

We deduce bounds for the partial derivatives on the whole range of under-approximations for $x$ and $y$ by

$$\boldsymbol{\Delta_i} = (\boldsymbol{\alpha_i^x \alpha_0^y} + \boldsymbol{\alpha_i^y \alpha_0^x}) + \sum_{j=1}^{i}(\boldsymbol{\alpha_i^x \alpha_j^y} + \boldsymbol{\alpha_i^y \alpha_j^x})\boldsymbol{\varepsilon_j} + \sum_{j=i+1}^{n}(\boldsymbol{\alpha_i^x \alpha_j^y} + \boldsymbol{\alpha_i^y \alpha_j^x})t_j. \quad (9)$$

Here, the coefficient $\boldsymbol{\alpha_i^z}$ is not always an over-approximation of $\frac{\partial z}{\partial \varepsilon_i}$.

*Practical considerations.* In both models, in order to obtain an under-approximation of the multiplication, the result of $\sum_{i=1}^{n} \boldsymbol{\Delta_i}(\boldsymbol{\varepsilon_i^*} - t_i)$ must be an improper interval. Considering that the $(\boldsymbol{\varepsilon_i^*} - t_i)$ are improper intervals containing zero, and therefore are in dual $\mathcal{Z}$, we then can deduce from table 1 what kind of intervals for $\boldsymbol{\Delta_i}$ lead to an improper interval for $\boldsymbol{\Delta_i}(\boldsymbol{\varepsilon_i^*} - t_i)$. The interval $\boldsymbol{\Delta_i}$ is proper, so it can be in $\mathcal{P}$, $-\mathcal{P}$ or $\mathcal{Z}$. If $\boldsymbol{\Delta_i} \in \mathcal{Z}$, then $\boldsymbol{\Delta_i}(\boldsymbol{\varepsilon_i^*} - t_i)$ is zero. Thus our interesting cases are $\boldsymbol{\Delta_i} \in \mathcal{P}$ or $\boldsymbol{\Delta_i} \in -\mathcal{P}$, which is satisfied when the $\boldsymbol{\Delta_i}$ intervals do not contain zero.

It is thus important to have the most accurate estimation of $\boldsymbol{\Delta_i}$ so that it does not include zero. Otherwise, a solution is to bisect one or several of the $\boldsymbol{\varepsilon_i}$ in such a way that on each bisection, our estimation for $\left\{\frac{\partial f^\varepsilon}{\partial \varepsilon_i}(\varepsilon),\ \varepsilon \in \text{pro } \boldsymbol{\varepsilon}\right\}$ does not contain zero. With model 2, we also have to find a trade-off between the width of $g^\varepsilon(t_1,\ldots,t_n)$ and the estimation of the corresponding $\boldsymbol{\Delta_i}$.

*Example 1.* Let us consider $f(x) = x^2 - x$ when $x \in [2,3]$. The interval of values taken by $f(x)$ is $[2,6]$. Here, an under-approximation of $f(x)$ can not be computed directly by Kaucher arithmetic, since variable $x$ does not appear only once in expression $f(x)$. An affine form for $x$ is $x = 2.5 + 0.5\varepsilon_1$, and we deduce

$$f^\varepsilon(\varepsilon_1) = (2.5 + 0.5\varepsilon_1)^2 - (2.5 + 0.5\varepsilon_1).$$

We bound the derivative by

$$\Delta_1 = 2 * 0.5 * (2.5 + 0.5\varepsilon_1) - 0.5 \subseteq [1.5, 2.5],$$

and, using the mean-value theorem with $t_1 = 0$, we have

$$\tilde{f}^\varepsilon(\varepsilon_1) = 3.75 + [1.5, 2.5]\varepsilon_1.$$

It can be interpreted as an under-approximation of the range of $f(x)$:

$$\check{\Gamma}(\tilde{f}^\varepsilon(\varepsilon_1)) = 3.75 + [1.5, 2.5][1, -1] = 3.75 + [1.5, -1.5] = [5.25, 4.25].$$

It can also be interpreted as an over-approximation:

$$\hat{\Gamma}(\tilde{f}^\varepsilon(\varepsilon_1)) = 3.75 + [1.5, 2.5][-1, 1] = 3.75 + [-2.5, 2.5] = [1.25, 6.25].$$

However, the range thus obtained is not as good as the one obtained by affine arithmetic as used in [9] for over-approximation, where $x^2 - x = [3.75, 4] + 2\varepsilon_1$, which gives the range $[1.75, 6]$.

*Example 2.* Consider $x = \frac{5}{2} + \frac{1}{2}\varepsilon_1$ and $y = \frac{9}{2} + \frac{1}{2}\varepsilon_2$. We compute an under-approximation, with model 1 for the multiplication and $(t_1, t_2) = (0, 0)$, and an over-approximation, with a semantics given in ([9]), of $z = y(x^2 - 2y)$, respectively noted as $\check{z}$ and $\hat{z}$:

$$\check{z} = -12.375 + [8, 15]\varepsilon_1 + [-8.125, -3.5]\varepsilon_2,$$

$$\hat{z} = -12.0625 + 11.25\varepsilon_1 - 5.8125\varepsilon_2 + 0.5625\varepsilon_3 + 1.5\varepsilon_4.$$

We obtain as estimates of the range, $[-23.875, -0.875] \sqsubseteq z \sqsubseteq [-31.1875, 7.0625]$. One bisection of $\varepsilon_1$ and $\varepsilon_2$ yields $[-28.453125, 2.765625] \sqsubseteq z \sqsubseteq [-31.1875, 5.8125]$. Two bisections again improve the estimation : $[-29.611328125, 3.689453125] \sqsubseteq z \sqsubseteq [-30.890625, 5.359375]$.

Using model 2 for the multiplication without bisection gives an improved result compared to model 1 without bisection:

$$\check{z} = -12.375 + [9, 13.5]\epsilon_1 + [-8.375, -3.375]\epsilon_2,$$

which concretizes as $[-24.75, 0] \sqsubseteq z$.

*Link between under and over-approximation.* We consider two variables $x$ and $y$, whose values are exactly described by affine forms with real coefficients (i.e. the under-approximation and over-approximation are equal), and we compute the multiplication $z = x \times y$.

Using model 1 with $(t_1, \ldots, t_n) = (0, \ldots, 0)$, we write

$$\check{z} = \alpha_0^x \alpha_0^y + \sum_{i=1}^{n} (\alpha_i^x \alpha_0^y + \alpha_i^y \alpha_0^x)\varepsilon_i + \left( \sum_{j=1}^{n} (\alpha_i^x \alpha_j^y + \alpha_i^y \alpha_j^x)\boldsymbol{\varepsilon_j} \right) \varepsilon_i. \qquad (10)$$

Computed with improper intervals for the $\varepsilon_i$, (10) gives an under-approximation of $z$. We saw that, computed with proper intervals for the $\varepsilon_i$, it gives an over-approximation, but better over-approximations can be obtained, as proposed in [9], as variations of

$$\hat{z} = \alpha_0^x \alpha_0^y + \sum_{i=1}^{n} (\alpha_i^x \alpha_0^y + \alpha_i^y \alpha_0^x)\varepsilon_i + (\sum_{i=1}^{n} |\alpha_i^x|.|\sum_{i=1}^{n} |\alpha_i^y|)\varepsilon_{n+1}. \qquad (11)$$

where a new noise symbol $\varepsilon_{n+1}$ is introduced to take into account the non affine part of the multiplication. We thus see that the part which is representable as an affine form of the existing symbols is shared between (10) and (11). In (10), the remaining part is expressed using the existing noise symbols $\varepsilon_i$ to $\varepsilon_n$, over-approximating existing relations. Whereas in (11), a new noise symbols is created. Thus all relations between the non-linear term and the other terms are lost, resulting in an over-approximation even if the range of this non linear term could be bounded precisely.

### 2.4 Order-Theoretic Considerations

Let in what follows $\check{x}$ and $\check{y}$ be under-approximating affine forms. Formally, we need to lift this domain of generalized affine forms so as to represent the empty set. Arithmetic operations are the lift of arithmetic operations defined in 2.

**Order.** We define the order by $\check{x} \sqsubseteq \check{y}$ if and only if $\forall i \geq 0$, $\boldsymbol{\alpha_i^x} \sqsubseteq \boldsymbol{\alpha_i^y}$. If $\check{x} \sqsubseteq \check{y}$, then we have on the concretization $\check{\Gamma}(\check{x}) \sqsubseteq \check{\Gamma}(\check{y})$.

In the case $\check{\Gamma}(\check{x})$ and $\check{\Gamma}(\check{y})$ are improper intervals and thus can be interpreted as under-approximations of the ranges of $x$ and $y$, this is equivalent to pro $\check{\Gamma}(\check{y}) \sqsubseteq$ pro $\check{\Gamma}(\check{x})$. Inclusion $\check{x} \sqsubseteq \check{y}$ thus expresses that $\check{x}$ is a better under-approximation that $\check{y}$, as it concretizes to an interval whose proper range is larger than the one of $\check{y}$.

*Example.* Let $\check{x} = 1 + [2,4]\varepsilon_1$ and $\check{y} = 1 + [1,5]\varepsilon_1$, so that $\check{x} \sqsubseteq \check{y}$. Using Kaucher arithmetic with improper $\boldsymbol{\varepsilon_1^*}$ and $\boldsymbol{\varepsilon_2^*}$, we compute $\check{\Gamma}(\check{x}) = 1 + [2,-2] = [3,-1]$ and $\check{\Gamma}(\check{y}) = 1 + [1,-1] = [2,0]$. We indeed have $[3,-1] \sqsubseteq [2,0]$, i.e. $[0,2] \sqsubseteq [-1,3]$.

This ensures even more: let $\check{C}$ be any mapping from program variables to affine forms (i.e. an abstract context), and let $e$ be any arithmetic expression. We denote by $\check{C}[z \leftarrow \check{x}]$ the context in which we replace the mapping for variable $z$ so as to get $\check{C}(z) = \check{x}$. We let $[\![e]\!]\check{C}$ denote the semantics of the arithmetic expression $e$ in context $\check{C}$ as defined in section 2.3. Then $\check{x} \sqsubseteq \check{y}$ implies, for all variables $z$,

$$\check{\Gamma}\left([\![e]\!]\check{C}[z \leftarrow \check{x}]\right) \sqsubseteq \check{\Gamma}\left([\![e]\!]\check{C}[z \leftarrow \check{y}]\right) \tag{12}$$

(analogous to the order relation for over-approximations defined in [9]), meaning that all future evaluations $e$ using $\check{x}$ will concretize to a bigger interval than using $\check{y}$. Hence, $\check{x}$, as an under-approximation, is more precise than $\check{y}$.

**Join.** The order-theoretic union is $\check{z} = \check{x} \cup \check{y}$, defined by

$$\check{z} = \check{x} \cup \check{y} = (\boldsymbol{\alpha_0^x} \cup \boldsymbol{\alpha_0^y}) + (\boldsymbol{\alpha_1^x} \cup \boldsymbol{\alpha_1^y})\varepsilon_1 + \ldots + (\boldsymbol{\alpha_n^x} \cup \boldsymbol{\alpha_n^y})\varepsilon_n.$$

One other solution is to take for $\check{z}$ either $\check{x}$ or $\check{y}$.

**Meet.** When, for all $i \geq 0$, $\boldsymbol{\alpha_i^x} \cap \boldsymbol{\alpha_i^y} \neq \emptyset$, we can define an under-approximation of the intersection by the order-theoretic intersection

$$\check{z} = \check{x} \cap \check{y} = (\boldsymbol{\alpha_0^x} \cap \boldsymbol{\alpha_0^y}) + (\boldsymbol{\alpha_1^x} \cap \boldsymbol{\alpha_1^y})\varepsilon_1 + \ldots + (\boldsymbol{\alpha_n^x} \cap \boldsymbol{\alpha_n^y})\varepsilon_n.$$

Otherwise, we can take the bottom element or enrich the abstract domain by propagating in further computations the over-approximated[1] constraints introduced on the values of the symbolic variables $\varepsilon$ :

$$(\boldsymbol{\alpha_0^x} - \boldsymbol{\alpha_0^y}) + (\boldsymbol{\alpha_1^x} - \boldsymbol{\alpha_1^y})\varepsilon_1 + \ldots + (\boldsymbol{\alpha_n^x} - \boldsymbol{\alpha_n^y})\varepsilon_n = 0.$$

[1] See the remark about the link with $\widetilde{pre}$ in section 2.4.

In practice, a set of interval constraints is attached to all affine forms, and a form of (interval) Gaussian elimination can be applied for normalizing the forms.

*Example.* Consider the following program, with independent inputs $x \in [-1, 3]$ and $b \in [2, 4]$:

```
x <- [-1,3]; b <- [2,5];
y = 2x + b;
if (y == x) s = 0;
else s = 1;
```

Interpreting the test `(y == x)` amounts to computing the intersection $x \cap y$. We have here the exact bounds for $x$ and $y$ (neither over-approximation nor under-approximation). With a computation in classical intervals, we have $y \in [0, 11]$, and we would find $s \in [0, 1]$. With affine forms, we have $x = 1 + 2\varepsilon_1$, $b = 3 + \varepsilon_2$, $y = 5 + 4\varepsilon_1 + \varepsilon_2$. For the intersection, we have to find values of $\varepsilon_1$ and $\varepsilon_2$ in $[-1, 1]$ satisfying constraint $5 + 4\varepsilon_1 + \varepsilon_2 = 1 + 2\varepsilon_1$. It simplifies to $\varepsilon_2 = -4 - 2\varepsilon_1$, with no solution. We deduce that the intersection is void, and $s = 1$.

**Widening.** A natural widening $\check{x}\nabla\check{y}$ is obtained using a widening on intervals

$$\check{x}\nabla\check{y} = (\boldsymbol{\alpha_0^x}\nabla\boldsymbol{\alpha_0^y}) + (\boldsymbol{\alpha_1^x}\nabla\boldsymbol{\alpha_1^y})\varepsilon_1 + \ldots + (\boldsymbol{\alpha_n^x}\nabla\boldsymbol{\alpha_n^y})\varepsilon_n.$$

We can define a narrowing similarly.

**Link with under-approximating abstractions.** Kaucher arithmetic provides a sound under-approximating abstract interpretation in the sense of [16], as we show now. Define as usual on intervals:

$$\begin{array}{rlcrl} \alpha^+ : \wp(\mathbb{R}) \rightarrow \mathbb{IR} & & \quad & \gamma^+ : & \mathbb{IR} \rightarrow \wp(\mathbb{R}) \\ S \rightarrow [\inf\ S, \sup\ S] & & & [a, b] \rightarrow & \{x \mid a \leq x \leq b\} \end{array}$$

and on improper intervals (using the notation $\wp(\mathbb{R})^{op}$ to denote the set of subsets of real numbers ordered with reverse inclusion $\subseteq^{op} = \supseteq$):

$$\begin{array}{rlcrl} \alpha^- : \wp(\mathbb{R}) \rightarrow \text{dual } \mathbb{IR} & & \quad & \gamma^- : \text{dual } \mathbb{IR} \rightarrow & \wp(\mathbb{R})^{op} \\ S \rightarrow [\sup\ S, \inf\ S] & & & [a, b] \rightarrow & \{x \mid a \geq x \geq b\} \end{array}$$

Of course, $(\alpha^+, \gamma^+)$ is the classical Galois connection for the interval abstraction. Now, whenever $\boldsymbol{i} \sqsubseteq \alpha^-(S^{op})$, we can prove that $\gamma^-(\boldsymbol{i}) \subseteq S^{op}$, hence $(\alpha^-, \gamma^-)$ is a dual Galois connection, hence under-approximating in the sense of [16]. Note that the logical interpretation of the four predicate transformers given in [16] is linked in the case of generalized interval arithmetic, to the logical interpretation of proper and improper intervals given in section 2.1: our forward under-approximating semantics for a functional $f$ is an abstraction of $post_f(S)$ (for $S$ a set of initial states). By the equality $post_f(S) = \widetilde{pre}_{f^{-1}}(S)$, which in turn is best under-approximated by $\widetilde{pre}_{f^{-1\sharp}}(S)$ ($f^{-1^\sharp}$ is the best over-approximation of the inverse of $f$, see [16]), we explain why in section 2.4 we needed to over-approximate the constraint to solve (by $\widetilde{pre}$) to get an under-approximation of the intersection with this constraint.

Note that the least fixed point in improper intervals, of a functional $F$ defining the abstract loop invariants, corresponds to the greatest fixed point of pro $F$, thus demonstrating that these under-approximating invariants are valid for all iterations of loops.

For affine forms, both over-approximating and under-approximating, we unfortunately do not have a best abstraction; hence correctness of our abstract semantics follows the generalized framework of [4]: for all variables $x$ of the program, the under-approximating form computed by our semantics is such that $S_x \subseteq^{op} \gamma^- \circ \check{\mathit{\Gamma}}(\check{x})$ in $\wp(\mathbb{R})^{op}$, where $S_x$ is the set of values that $x$ can take. In fact, letting $\llbracket e \rrbracket_c$ stand for the concrete semantics of a term $e$,

$$\check{\mathit{\Gamma}}\left(\llbracket e \rrbracket \check{C}\right) \sqsubseteq \llbracket e \rrbracket_c \gamma^- \circ \check{\mathit{\Gamma}}(\check{C}).$$

All further evaluations of a set of under-approximating affine forms will give an under-approximation of the real set of results. In particular, the dependencies are well encoded in the semantics. Note also that one can replace, for any variable $x$, $\check{x}$ by any $\check{x}'$ with $\check{x} \sqsubseteq \check{x}'$ and the same property will hold with the newly defined context thanks to (12).

### 2.5 Complexity

The number of terms of the generalized affine forms resulting from our under-approximating semantics for a given program, is bounded by the number of uncertain inputs of this program. By uncertain inputs we mean the number of inputs which value is not known exactly but given in an interval of values. Of course, this is a pessimistic upper bound : all variables will not depend on all these uncertain inputs.

The addition, subtraction, join, meet, and widening, are linear in the number of terms of the affine forms; the multiplication is quadratic in this number of terms. Of course, these complexity results suppose we do not bisect the $\varepsilon_i$ variables.

## 3 Applications and Experiments

### 3.1 Implementation

A C library implementing the different under-approximating semantics of this paper has been implemented. Of course, it does not have access to exact real arithmetic. However, computing the interval coefficients $\boldsymbol{\alpha}_{\boldsymbol{i}}^{\boldsymbol{x}}$ using outer rounding (i.e. rounding towards $+\infty$ for the upper bound of the interval and towards $-\infty$ for the lower bound) ensures correctness of the result. Also, using a multiple precision library such as MPFR to compute the bounds of these intervals can improve the accuracy.

### 3.2 Combination of Under and Over-Approximations

We consider, for some given $A$, the (non-linear) iteration of the Newton algorithm $x_{i+1} = 2x_i - Ax_i^2$. If we take $x_0$ not too far away from the inverse of $A$, this iteration converges to the inverse of $A$. As an example, we take $A \in [1.99, 2.00]$, i.e. $\check{A} = 1.995 + .005\epsilon_1$, $x_0 = .4$ and ask to iterate this scheme until $|x_{i+1} - x_i| < 5e^{-6}$. We get the concretizations of lower and upper forms shown in the left part of figure 1. After 6 iterations, we get stable concretizations, for both lower and over-approximations of $x_6$ :

$$[0.5012531328, 0.5012531328] \sqsubseteq x_6 \sqsubseteq [0.499996841, 0.502512574],$$

$$[0, 0] \sqsubseteq |x_6 - x_5| \sqsubseteq [-3.17241289e^{-6}, 3.17241289e^{-6}].$$

And for the stopping criterion $|x_{i+1} - x_i| < 5e^{-6}$, using simultaneously the over- and under-approximation, we obtain that the Newton algorithm terminates after exactly 4 iterations, with

$$[0.5002532337, 0.5022530319] \sqsubseteq x_4 \sqsubseteq [0.499996841, 0.502512574].$$

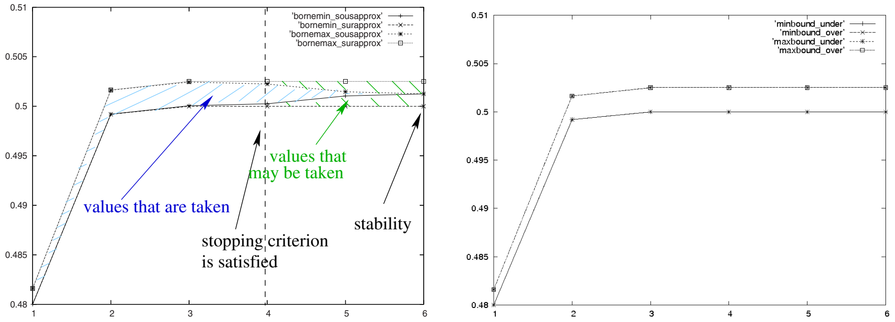
Of course, this is a general fact: the combination of under and over approximations gives in general a very powerful method to determine the invariants of a program.

We can also refine the results using subdivisions of the input $A$, that give way to as many independent computation relying on different affine forms for A. For example, if we subdivide this initial interval in 2 sub-intervals, we get

$$[0.50006320027, 0.50244772292] \sqsubseteq x_4 \sqsubseteq [0.49999370736, 0.50251570685],$$

$$[0.50062578222, 0.50188205771] \sqsubseteq x_6 \sqsubseteq [0.49999370676, 0.50251570746].$$

And when subdividing in 32 sub-intervals, we get after 6 iterations a relative difference between the bounds given by the over- and under-approximations, of less than $9e^{-6}$; the results obtained are also shown in the right part of figure 1.



no subdivision

32 subdivisions

**Fig. 1.** Estimation of the maximum value of result $x_i$ in the Newton algorithm

### 3.3 Filters, Perturbations and Worst Case Scenario

In the sequel, in order not to get into complicated details, we suppose that we have a real arithmetic at hand (or arbitrary precision arithmetic).

**Linear scheme.** Consider the following filter of order 2:

$$S_i = 0.7E_i - 1.3E_{i-1} + 1.1E_{i-2} + 1.4S_{i-1} - 0.7S_{i-2},$$

where $E_i$ are independent inputs between 0 and 1, so that $\check{E}_i = \hat{E}_i = \frac{1}{2} + \frac{1}{2}\epsilon_{i+1}$, and $S_0 = S_1 = 0$. We first consider the output $S_i$ of this filter for a fixed number of unfoldings, e.g. $i = 99$. Using the over-approximating semantics of [9], our prototype gives us

$$\hat{S}_{99} = \check{S}_{99} = 0.83 + 7.81e^{-9}\varepsilon_1 - 2.1e^{-8}\varepsilon_2 - 1.58e^{-8}\varepsilon_3 + \ldots - 0.16\varepsilon_{99} + 0.35\varepsilon_{100};$$

whose concretization gives an exact (under the assumption that the coefficients of the affine form are computed with arbitrary precision) enclosure of $S_{99}$:

$$[-1.0907188500, 2.7573854753] \sqsubseteq S_{99} \sqsubseteq [-1.0907188500, 2.7573854753].$$

Also, the affine form gives the sequence of inputs $E_i$ that maximizes $S_{99}$ : $E_i = 1$ if the corresponding coefficient multiplying $\varepsilon_{i+1}$ is positive, $E_i = -1$ otherwise.

Note that the exact enclosure actually converges to

$$S_\infty = [-1.09071884989..., 2.75738551656...],$$

and therefore the signal (sequence of inputs of size 99) leading to the maximal value of $S_{99}$ is a very good estimate of the signal leading to the maximal value of $S_i$, for any $i \geq 99$. This exact enclosure is given as the limit of the concretisation of the over-approximating iterative scheme (fully unfolded), which stabilizes in finite time for a fixed precision arithmetic, with correct outer rounding (here in `double` precision). It is to be noted that the over-approximating domain of [9], slightly improved (will be published elsewhere), does find a finite over-approximation of these bounds, but this is not the subject of this article.

This can be used to find bad-case scenarios of a program : for a specification of the filter forbidding an output greater than 2.5, this sequence of inputs provides a counter-example.

This generalizes to linear recursive filters of any order:

**Lemma 1.** *Consider a linear recursive filter of order n:*

$$s_n = \sum_{k=0}^{N-1} a_k s_{n-k-1} + \sum_{k=0}^{N} b_k e_{n-k}$$

*where $s_k$ is the output at iterate $k$, and $e_k$ is the input at iterate $k$. Then:*

*(1) When unfolding k times, under and over approximating forms are equal, and their concretization gives the exact range for $s_k$, up to rounding errors due to the implementation of the abstract domains*
*(2) The under-approximating form after k unfoldings provides the sequence of inputs that lead to the maximum range of the kth output.*
*(3) When the filter is stable, we can make the under-approximation of the output arbitrarily close to the real range $s_\infty$, by unfolding k times for large enough k.*

**Perturbation by a non linear term.** We now perturb this linear scheme by adding a non-linear term $0.005E_iE_{i-1}$, obtaining

$$S_i = 0.7E_i - 1.3E_{i-1} + 1.1E_{i-2} + 1.4S_{i-1} - 0.7S_{i-2} + 0.005E_iE_{i-1}.$$

Again, we analyze $S_i$ for a fixed number $i = 99$ of unfoldings. Using the over-approximating semantics of [9], we get

$$\hat{S}_{99} = 0.837 + 7.81e^{-9}\varepsilon_1 - 2.09e^{-8}\varepsilon_2 - 1.58e^{-8}\varepsilon_3 + \ldots - 0.157\varepsilon_{99} + 0.351\varepsilon_{100} \\ +1.77e^{-11}\varepsilon_{101} - 2.66e^{-11}\varepsilon_{102} + \ldots + 0.00175\varepsilon_{197} + 0.00125\varepsilon_{198},$$

in which terms from $\varepsilon_{101}$ to $\varepsilon_{198}$ account for the over-approximation of non-linear computations, and do not correspond to inputs. A sequence of inputs leading to a bad-case scenario is thus not given directly by the sign of the $\varepsilon_1, \ldots, \varepsilon_{100}$ as in the linear case. Hence one cannot use the same technique as before, to be sure to reach the supremum of the range for $S_{99}$. One can get a plausible worst-case scenario by choosing the $E_0, \ldots, E_{99}$ that maximize the sub affine form containing only these $\epsilon_k$, but one has no assurance that this might be even close to the real supremum of $S_{99}$. But we will show that the under-approximating form allows us to choose at least part of the inputs.

Using the under-approximating semantics that we have developed in this paper, and model 2 for the multiplication, we get:

$$\check{S}_{99} = 0.837 + 7.81e^{-9}\varepsilon_1 - 2.09e^{-8}\varepsilon_2 + \ldots + [-0.0577, 0.0635]\varepsilon_{93} \\ +[0.0705, 0.138]\varepsilon_{94} + [0.185, 0.223]\varepsilon_{95} + [0.25, 0.271]\varepsilon_{96} + [0.222, 0.234]\varepsilon_{97} \\ +[0.081, 0.0876]\varepsilon_{98} + [-0.158, -0.155]\varepsilon_{99} + [0.35, 0.352]\varepsilon_{100}.$$

This gives the following estimates for the real enclosure of $S_{99}$:

$$[-0.47655194955570, 2.1515519079] \sqsubseteq S_{99} \sqsubseteq [-1.10177396494, 2.77677392330].$$

Using model 1 for the multiplication, we get the slightly less precise under-approximation $[-0.435, 2.11] \sqsubseteq S_{99}$. But, when the interval coefficient $\boldsymbol{\alpha_k}$ corresponding to $\varepsilon_k$ does not contain zero, we know what is the good choice of input $E_{k-1}$ (see Lemma 2). This cannot be proved for the form obtained with model 2 of the multiplication. However in the general case it remains a good heuristic. And in the particular case here, the interval coefficients have the same signs for the two forms. We thus know that $E_{93} = 1$ - note that $E_i$ corresponds to $\varepsilon_{i+1}$! - $E_{94} = 1$, $E_{95} = 1$, $E_{96} = 1$, $E_{97} = 1$, $E_{98} = 0$ and $E_{99} = 1$ is the best depth 7 choice of inputs that will maximize $S_{99}$. In order to get an estimate of the supremum for $S_{99}$, one can try any inputs for $E_0$ to $E_{92}$. Inputs $E_0, \ldots, E_{92} = 0$ give, for instance, $S_{99} = 2.460374$. As a heuristic, one can use the $\varepsilon_0, \ldots, \varepsilon_{92}$ that maximize the over-approximating term, giving $S_{99} = 2.766383$. A one hour simulation on a 2GHz PC for $10^9$ random sequences of 100 entries gives as estimate of the supremum 2.211418, trailing our estimate in both time and precision.

We can generalize again:

**Lemma 2.** *Suppose we have $\check{x} = \boldsymbol{\alpha_0^x} + \sum_{i=1}^{n} \boldsymbol{\alpha_i^x} \varepsilon_i$ (with model 1 of multiplication), giving an under-approximation for $x = f(\varepsilon_1, \ldots, \varepsilon_n)$ on $[-1,1]^n$ with $f$ continuous, and $I_+ = \{i \mid 1 \leq i \leq n,\ \alpha_i \in \mathcal{P}\}$, $I_- = \{i \mid 1 \leq i \leq n,\ \alpha_i \in -\mathcal{P}\}$, $I_z = \{i \mid 1 \leq i \leq n,\ \alpha_i \in \mathcal{Z}\}$. Then the supremum of $f$ on $[-1,1]^n$ is reached for some set of values $\varepsilon_1, \ldots, \varepsilon_n \in [-1,1]$ with $\epsilon_i = 1$ for $i \in I_+$, $\epsilon_i = -1$ for $i \in I_-$. A similar result holds for the infimum of $f$.*

And as with the linear scheme, the under and over-approximations converge towards the following estimates, very close to the estimate of $S_{99}$ :

$$[-0.4765519, 2.1515519] \sqsubseteq S_\infty \sqsubseteq [-1.10177396500, 2.77677396500].$$

Hence the previously found signal gives a scenario which is very close to the worst-case scenario. Indeed $S_{99} = 2.766383$ for which we found a scenario with our heuristics cannot be more than half a percent away from the true maximum.

## 4 Conclusion and Related Work

We have shown how to give a practical, tractable abstract semantics for under-approximating the values of program variables. Combined with an over-approximating analysis such as the one of [9], it gives a good indication of the quality of the static analysis performed. And the combination can sometimes be used to improve the analysis results, as shown in section 3.2. We can also, as side products of this analysis, give good estimations of worst-case scenarios, that lead to maximal or minimal values of some variable.

This under-approximating abstract semantics is for the time being applied to real-valued variables, and does not address yet floating-point variables. Indeed, floating-point numbers are discrete, hence no convex under-approximation within real numbers other than a single point or empty set can be found. One can think of adding information about the minimal size of the gaps between two floating-point values in the resulting interval. We hope to be able to have similar results on worst-case scenarios in that context, in particular for producing executions which maximize the imprecision error. This is left for future work.

Another direction we pursued was to extend the method of this paper to higher-order Taylor forms. We can indeed give a semantics based on a Taylor expansion of arbitrary degree to any program. But we do not know yet how to conclude, except in particular cases, on under-approximated bounds, contrarily to the case of over-approximations (see for instance [1] for a similar observation on over-approximations and Taylor forms).

Finally, the order-theoretic join and meet rely directly on intervals, it is hence most probable that policy iteration techniques [2] can be used on this domain.

## References

1. Chapoutot, A., Martel, M.: Différentiation automatique et formes de Taylor en analyse statique de programmes numériques (in French). In: AFADL'07 (2007)
2. Costan, A., Gaubert, S., Goubault, E., Martel, M., Putot, S.: A policy iteration algorithm for computing fixed points in static analysis of programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, Springer, Heidelberg (2005)
3. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points. Principles of Programming Languages 4, 238–252 (1977)
4. Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation 2(4), 511–547 (1992)
5. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL'78, pp. 84–97 (1978)
6. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM Trans. Prog. Lang. Systems 19, 253–291 (1997)
7. Goldsztejn, A.: Modal intervals revisited. Submitted to Reliable Computing
8. Goldsztejn, A., Daney, D., Rueher, M., Taillibert, P.: Modal intervals revisited: a mean-value extension to generalized intervals. In: QCP'05 (2005)
9. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 18–34. Springer, Heidelberg (2006)
10. Grumberg, O., Lerda, F., Strichman, O., Theobald, M.: Proof-guided underapproximation-widening for multi-process systems. In: POPL (2005)
11. Kaucher, E.W.: Interval analysis in the extended interval space IR. Computing (Supplementum) 2, 33–49 (1980)
12. Kaucher, E.W.: Uber metrische und algebraische eigenshaften einiger beim numerischen rechnen auftretender raume, PhD thesis, Karlsruhe (1973)
13. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001)
14. Pasareanu, C.S., Pelánek, R., Visser, W.: Concrete model checking with abstract matching and refinement. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 52–66. Springer, Heidelberg (2005)
15. Schmidt, D.A.: A calculus of logical relations for over- and underapproximating static analyses. Sci. Comput. Program. 64(1), 29–53 (2007)
16. Schmidt, D.A.: Underapproximating predicate transformers. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 127–143. Springer, Heidelberg (2006)
17. Stolfi, J., de Figueiredo, L.H.: An introduction to affine arithmetic, TEMA (2003)

# A Framework for End-to-End Verification and Evaluation of Register Allocators

V. Krishna Nandivada[1], Fernando Magno Quintão Pereira[2], and Jens Palsberg[2]

[1] IBM India Research Laboratory, Delhi
[2] UCLA Computer Science Department, University of California, Los Angeles

**Abstract.** This paper presents a framework for designing, verifying, and evaluating register allocation algorithms. The proposed framework has three main components. The first component is MIRA, a language for describing programs prior to register allocation. The second component is FORD, a language that describes the results produced by the register allocator. The third component is a type checker for the output of a register allocator which helps to find bugs. To illustrate the effectiveness of the framework, we present RALF, a tool that allows a register allocator to be integrated into the gcc compiler for the StrongARM architecture. RALF simplifies the development of register allocators by sheltering the programmer from the internal complexity of gcc. MIRA and FORD's features are sufficient to implement most of the register allocators currently in use and are independent of any particular register allocation algorithm or compiler. To demonstrate the generality of our framework, we have used RALF to evaluate eight different register allocators, including iterated register coalescing, linear scan, a chordal based allocator, and two integer linear programming approaches.

## 1 Introduction

### 1.1 Background

The register allocator is one of the most important parts of a compiler. Our experiments show that an optimal algorithm can improve the execution time of the compiled code by up to 250%. Although researchers have studied register allocation for a long time, many interesting problems remain. For example, in recent years PLDI (ACM SIGPLAN Conference on Programming Language Design and Implementation) has published several papers on register allocation [2004 (2 papers), 2005 (3 papers), 2006 (2 papers)]. While the essence of register allocation is well understood, developing a high-quality register allocator is nontrivial. In addition to understanding the register allocation algorithm, which can be complex, the developer must also know the internals of the compiler where the allocator will be implemented. For example, public domain compilers such as GCC [2] or SMLNJ [1] and compiler frameworks such as SUIF [15] or SOOT [31] allow a programmer to implement a new register allocator. However, the programmer has to understand and work with their data structures,

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 153–169, 2007.

which are complicated because register allocation affects both the machine specific and machine independent parts of the compilation process. One attempt to address this problem was Tabatabai et al's. [3] register allocation framework, implemented in the CMU C compiler. Their framework presents modules (for example, graph construction, coalescing, color assignment, spill code insertion, and others) that different register allocators might need. However, if the allocator needs mechanisms other than those provided by the framework, the programmer must still deal with the internals of the CMU C compiler. A goal of our work is to completely shield the developer of register allocators from the internal complexities of a compiler.

Debugging register allocators is also a complicated task. Errors may surface in non-trivial ways; sometimes many instructions after the incorrect code. Moreover, the low-level nature of the machine code and its large size makes visual inspection of the register-allocator's output tedious and error-prone. As a testimony of these difficulties, most recent publications in this field report only static data and not run-time measurements, let alone implementations in industrial compilers. Although static data (such as number of spills, and number of registers used) is important, it does not reflect the behavior of the register allocator in the presence of other optimizations and run time factors.

A few researchers have developed techniques for proving register allocators correct. Naik and Palsberg [21] proved the correctness of the ILP-based register allocator of Appel and George [6]. Ohori [26] designed a register allocation algorithm as a series of proof transformations which is correct by construction. Other researchers have shown how to validate the output of register allocators. Necula [24] presented a translation validation infrastructure for the gcc compiler that includes register allocation. Necula's scheme treats the memory address of a spilled register as a variable, which allows reasoning about its live ranges, although relying on specific characteristics of the gcc compiler, such as addressing modes. Leroy [17] formally describes a technique to validate the output of graph coloring based register allocation algorithms. Basically, if the interference graph contains a pair of adjacent temporaries allocated to the same register, the verifier emits an error, otherwise it assumes that the code generated is correct. Andersson [5] and Pereira et al. [27] adopted similar approaches. Huang et al. [16] presented a more general approach which matches the live ranges of values in the original program against the live ranges of machine locations in the register-allocated program. A goal of our work is to use a type system to validate the output of register allocators, and to prove the soundness of the type system itself.

Annotations in the register-allocated code can help validation algorithms. Morrisett et al. [20] used type annotations to help guarantee memory safety, Necula and Lee [25] used more general annotations such as memory bounds, and Agat [4] used type annotations to validate the output of a register allocator. A goal of our work is to validate the output of register allocators without extra annotations in the target code.

### 1.2 Our Results

We present a framework for designing, verifying, and evaluating register allocators. Our framework shields the developer from the internal complexities of a compiler, uses a type system to validate the output of register allocators, and does not rely on code annotations.

**MIRA and FORD.** Our framework centers around MIRA (Mathematical Intermediate representation for Register Allocation), a language for describing programs prior to register allocation, and FORD (FOrmat for Register allocation Directives), a language that describes the results produced by the register allocator. MIRA sources are abstract intermediate representations of programs immediately before the register allocation phase. MIRA descriptions contain architecture and program specific information. The former includes information such as number and classes of machine registers, number of caller-save registers, and costs of loads and stores. The latter consists of information such as the program's control flow graph, use and definition sites of each variable, and estimated usage frequency of each instruction. In the context of our framework, the register allocator emits FORD directives that control spill code generation, register and variable mapping at different program points, and any additional code that needs to be inserted (For example, move instructions). MIRA and FORD can accommodate many of the traditional register allocation algorithms and are simple enough to be easily used by the developer of register allocators.

**Type system.** We use a type system to verify that the output of a register allocator is correct. Our type system was inspired by Morrisett et al.'s type system for assembly language [20] and can be used with intermediate representations other than MIRA and FORD. A type correct program is guaranteed to have properties such as: (1) pseudos whose live ranges overlap are assigned to different registers, (2) live ranges of the same pseudo always reach a join point assigned to the same register, and (3) a live register is not overwritten before it is used. We have found that typical errors in the implementation of a register allocator violate these properties. The type checker points out the locations of the register-allocated code where these properties fail. We have proved type soundness for our type system using the Twelf Meta-theorem prover [32].

**RALF.** Our tool RALF (Register ALlocation Framework) allows a programmer to plug a new register allocator into gcc, without requiring the programmer to know any details of gcc's implementation. RALF is an extra layer added on top of gcc, acting as a glue between gcc and the plugged-in register allocator. The new register allocator takes a MIRA program as input and gives a collection of FORD directives as output. The main objective of RALF is to be *simple*: when writing a register allocator compatible with our framework, the developer only has to write a program that translates MIRA to FORD. RALF treats the register allocator, which can be implemented in any language, as a black box whose only purpose is to translate MIRA sources (provided by RALF) to FORD directives (which are fed back to RALF). Given an input program and the plugged-in register allocator, RALF generates a StrongARM binary executable using the new register allocator and the rest of gcc. RALF uses our type checker to verify the output of the register allocator. In addition to our own experiments with RALF, we have used RALF in an advanced compiler course at UCLA. As part of a class assignment, each student implemented a different register allocation algorithm to be used with the framework. More about these experiences can be found at RALF's homepage `http://compilers.cs.ucla.edu/ralf`. Our current implementation of RALF compiles only C code to the StrongARM architecture; however, the MIRA and FORD description languages are designed to be general enough to fit other source languages and architectures. The RISC nature of the StrongARM architecture helps to keep RALF simple. Our implementation, which gets activated by different compiler switches, contains around 5000 lines of C code divided among 125 functions. The proposed framework is not intended for industrial implementations of register allocators, but for fast implementation and testing of research prototypes. The techniques used in our type system can be used also in a production compiler.

**Comparison of eight register allocators.** We have used RALF to implement and compare eight different registers allocators. The allocators tested range from classical algorithms, such as the usage-count based implementation [11], to novel approaches, such as register allocation via coloring of chordal graphs [27]. In addition of using static data, such as number of variables spilled, we compare the different algorithms by running the produced code on a StrongARM processor.

The remainder of this paper is organized as follows: Section 2 describes a simplified version of MIRA and FORD and characterizes the register allocation problem. Section 3 presents our type system, and Section 4 discusses our experimental results.

## 2 A Simplified View of MIRA and FORD

Register allocation is the process of mapping a program $\mathcal{M}$ that can use an unbounded number of variables, or *pseudo-registers*, to a program $\mathcal{F}$ that must use a fixed (and generally small) number of *machine registers* to store data. In the remainder of this paper we use *register* in place of *machine register*, and *pseudo* for *pseudo-register*. If the number of registers is not sufficient to accommodate all the pseudos, some of them must be stored in memory; these are called *spilled* pseudos. Following the nomenclature normally used in the gcc community, we call the intermediate representation of programs $\mathcal{M}$ the *Register Transfer Language (RTL)* and we use *Location Transfer Language (LTL)* to describe programs $\mathcal{F}$. As shown in Figure 1, MIRA and FORD have been designed to constitute an interface between the register allocator and the RTL/LTL intermediate representations. They facilitate the development of register allocation algorithms by hiding from the algorithm's designer the details of the RTL/LTL representation that are not relevant to the register allocation process. In order to precisely characterize the register allocation problem, we will use a simplified version of MIRA and a simplified version of FORD in this section. We will call them sMIRA and sFORD respectively. For our purposes, a register allocator *RA* is a black box

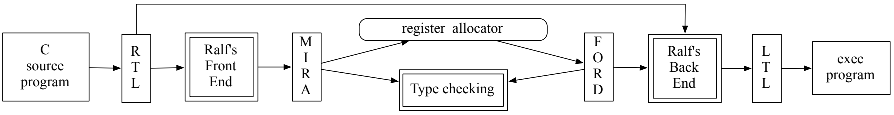


**Fig. 1.** Block diagram of our register allocation framework

which takes an sMIRA program $P_{sm}$ and a description of the target architecture, and produces an sFORD program $P_{sf}$. In this section, we will assume that the architecture specific information is a list of $K$ machine registers $\mathsf{Regs}$, and a set of caller save registers $\mathsf{CallerSave} \subseteq \mathsf{Regs}$. Thus, $P_{sf} = RA(P_{sm}, \mathsf{Regs}, \mathsf{CallerSave})$. An actual register allocator would require more information, such as the class of each machine register, the cost of different operations, etc. Such information is present in the concrete specification of MIRA and FORD, which is given in the full version of this paper, available at `http://compiler.cs.ucla.edu/ralf`.

### 2.1 Simplified MIRA

sMIRA programs are described by the grammar in Figure 2(a). We adopt an abstract representation of programs. All the operands not relevant to register allocation, such as constants, heap memory addresses, and others, are represented with the symbol •. The only explicit operands are pseudos ($p$) and pre-colored registers $(r, p)$. The latter are pseudos that have been assigned a fixed machine register due to architectural constraints. In this simplified presentation, we only use pre-colored register to pass parameters to function calls, and to retrieve their return value. Other operands, such as constants and memory references on the heap are abstracted out.

An sMIRA program is a sequence of *instruction blocks*. Each instruction block consists of an address label, represented by $L$, heading a sequence of instructions ($I$) followed by a jump. sMIRA programs can use an unbounded number of pseudo registers $p$. We do not distinguish the opcode of instructions, except *branches* and function calls. Branches affect the control flow, and *function calls* may cause caller save registers to be overwritten, once register allocation has been performed. A function call such as $(r_0, p_0) = \mathsf{call}\ (r_1, p_1)..(r_s, p_s)$ uses pre-colored pseudos $(r_1, p_1)..(r_s, p_s)$ as parameters, and produces a return value in the pre-colored pseudo $(r_0, p_0)$. Notice that in sMIRA a call instruction does not contain a label; that is because we support only intra-procedural register allocation. An example of sMIRA program is given in Figure 2(b).

### 2.2 Simplified FORD

A register allocator produces sFORD programs, which are represented by the grammar in Figure 3(a). Operands in sFORD are bindings of pseudos ($p$) to *machine locations*. In our representation, a machine location can be either a

| | |
|---|---|
| (Programs) | $P_{sm} ::= L_1 I_1; \ldots ; L_k I_k$ |
| (Code labels) | $L ::= L_1 \mid L_2 \mid \ldots$ |
| (Instr. Sequence) | $I ::=$ |
| - (Jump) | $\mid$ jump $L$ |
| - (Sequence) | $\mid i; I$ |
| (Pseudos) | $p ::= p_1 \mid p_2 \mid \ldots$ |
| (Registers) | $r ::= r_1 \mid \ldots \mid r_k$ |
| (Operands) | $o ::=$ |
| - (Constants) | $\mid \bullet$ |
| - (Pseudos) | $\mid p$ |
| - (Pre-coloreds) | $\mid (r, p)$ |
| (Instructions) | $i ::=$ |
| - (Assignment) | $\mid p = o$ |
| - (Ass. pre-col.) | $\mid (r, p) = o$ |
| - (Cond. jump) | $\mid$ if $p$ jump $L$ |
| - (Function call) | $\mid (r_0, p_0) =$ call $(r_1, p_1)..(r_s, p_s)$ |

**(a)**

| | |
|---|---|
| $L_1$ | $L_2$ |
| $p_0 = \bullet$ | $p_0 = p_0$ |
| $p_1 = \bullet$ | $(r_0, p_6) = \bullet$ |
| $p_2 = p_0$ | $(r_0, p_6) =$ call $(r_0, p_6)$ |
| if $p1$ jump $L_3$ | $p_2 = (r_0, p_6)$ |
| jump $L_2$ | $p_3 = p_0$ |
| | $p_4 = \bullet$ |
| $L_3$ | $p_7 = p_4$ |
| $p_4 = p_0$ | if $p_3$ jump $L_2$ |
| $p_5 = p_2$ | jump $L_3$ |
| jump $exit$ | |

**(b)**

**Fig. 2.** (a) Syntax of sMIRA programs (b) Example of sMIRA program

| | |
|---|---|
| (Programs) | $P_{sf} ::= L_1\ I_1; \ldots ; L_k\ I_k$ |
| (Code Labels) | $L ::= L_1 \mid L_2 \mid \ldots$ |
| (Inst. seq.) | $I ::=$ |
| - (Jump) | $\mid$ jump $l$ |
| - (Sequence) | $\mid i; I$ |
| (Pseudos) | $p ::= p_1 \mid p_2 \mid \ldots$ |
| (Registers) | $r ::= r_1 \mid r_2 \mid \ldots$ |
| (Mem. locs.) | $l ::= l_1 \mid l_2 \mid \ldots$ |
| (Operands) | $o ::=$ |
| - (Constant) | $\mid \bullet$ |
| - (Reg. bind) | $\mid (r, p)$ |
| - (Mem. bind) | $\mid (l, p)$ |
| (Instructions) | $i ::=$ |
| - (Assig.) | $\mid (r, p) = o$ |
| - (Store) | $\mid (l, p) = o$ |
| - (Cond. jump) | $\mid$ if $(r, p)$ jump $L$ |
| - (Func. call) | $\mid (r_0, p_0) =$ call $(r_1, p_1)..(r_s, p_s)$ |

**(a)**

| | |
|---|---|
| $L_1$ | $L_2$ |
| $(r_1, p_0) = \bullet$ | $(r_1, p_0) = (r_1, p_0)$ |
| $(r_0, p_1) = \bullet$ | $(r_0, p_6) = \bullet$ |
| $(r_1, p_2) = (r_1, p_0)$ | $(l_1, p_0) = (r_1, p_0)$ |
| $(l_0, p_2) = (r_1, p_2)$ | $(r_0, p_6) =$ call $(r_0, p_6)$ |
| if $(r_1, p1)$ jump $L_3$ | $(r_1, p_0) = (l_1, p_0)$ |
| jump $L_2$ | $(r_2, p_2) = (r_0, p_6)$ |
| | $(l_0, p_2) = (r_2, p_2)$ |
| $L_3$ | $(r_2, p_3) = (r_1, p_0)$ |
| $(r_1, p_4) = (r_1, p_0)$ | $(r_0, p_4) = \bullet$ |
| $(r_0, p_2) = (l_0, p_2)$ | $(r_0, p_7) = (r_0, p_4)$ |
| $(r_0, p_5) = (r_0, p_2)$ | if $(r_1, p_3)$ jump $L_2$ |
| jump $exit$ | jump $L_3$ |

**(b)**

**Fig. 3.** (a) Syntax of sFORD programs (b) Example of sFORD program physical register ($r$), or a memory address ($l$). In addition to calls and branches, we distinguish loads "$= (l, p)$", and stores "$(l, p) =$", because these instructions are used to save and restore spilled values. Notice that we make an explicit distinction between *code labels*, represented by $L$, and data labels (stack locations), represented by $l$. In the sFORD representation, caller-save registers can be overwritten by function calls; thus, the register allocator must guarantee that pseudos that are alive across function calls are not mapped to caller-save registers.

Let $P_{sm}$ be the sMIRA program in Figure 2(b), and consider an architecture where $\mathsf{Regs} = \{r_0, r_1, r_2\}$ and $\mathsf{CallerSave} = \{r_0, r_1\}$. Let $RA$ be a hypothetical register allocator, such that $P_{sf} = RA(P_{sm}, \mathsf{Regs}, \mathsf{CallerSave})$ is the program in Figure 3(b). In our example, $RA$ has allocated register $r_1$ to pseudo $p_0$ in the first instruction of $L_1$. The pseudo $p_2$ has been spilled due to the high register pressure in block $L_2$; its memory location is given by the label $l_0$. Furthermore, pseudo $p_0$ has been spilled to memory location $l_1$ because it is stored in the caller save register $r_0$ and is alive across a function call.

## 3 Type Checking

Inaccuracies in the implementation of a register allocation algorithm may result in different types of errors in the sFORD program. In Figure 4 we illustrate five different errors that can be produced by a flawed register allocator. In Fig. 4(a), $p_0$ was defined in register $r_0$ at instruction 1, but it is expected to be found in register $r_1$ when used in instruction 2. In Fig. 4(b) register $r_0$ is overwritten in instruction 2 while it contains the live pseudo $p_0$. Fig. 4(c) describes a similar situation, but in this case a memory location is overwritten while the value it holds is still alive. In Fig. 4(d), we assume that $r_1$ is a caller save register. In this case, pseudo $p_1$ may have its location overwritten during the execution of the function call in instruction 2. Finally, in Fig. 4(e) the value of $p_0$, stored in register $r_0$ may be overwritten, depending on the path taken during the execution of the program. This last error is particularly elusive, because its consequences might not surface during the testing of the target program.

In order to guarantee that the values used in the sMIRA program are preserved in the sFORD representation, we use a type checker inspired by [20]. The basic data used in our type system are machine locations, and the type of a machine location is the pseudo-register that it stores. In our case, the register allocator

| | | | | |
|---|---|---|---|---|
| $(r_0, p_0) = \bullet$<br>$(r_2, p_2) = (r_1, p_0)$ | $(r_0, p_0) = \bullet$<br>$(r_0, p_1) = \bullet$<br>$(r_1, p_2) = (r_0, p_0)$ | $(l_0, p_0) = (r_0, p_0)$<br>$(l_0, p_1) = (r_1, p_1)$<br>$(r_1, p_2) = (l_0, p_0)$ | $(r_1, p_1) = \bullet$<br>$(r_0, p_0) = \mathsf{call}$<br>$(r_2, p_2) = (r_1, p_1)$ | $L_1$ $(r_0, p_0) = \bullet$<br>$\mathsf{if}$ $(r_0, p_0)$ $\mathsf{jump}$ $L_2$<br>$(r_0, p_1) = \bullet$<br>$\mathsf{jump}$ $L_2$<br>$L_2$ $(r_1, p_2) = (r_0, p_0)$ |
| **(a)** | **(b)** | **(c)** | **(d)** | **(e)** |

**Fig. 4.** (a-e) Examples of errors due to wrong register allocation

annotates each definition or use of data with its type. A definition of a machine location, e.g $(r_i, p_j) = \bullet$ corresponds to declaring $r_i$ with the type $p_j$. Let $(r_i, p_j)$ be an annotated machine location. Intuitively, every time this machine location is used, e.g $(r, p) = (r_i, p_j)$, its annotated type corresponds to the type that can be discovered by a type inference engine if the sFORD program is correct. For instance, the program in Figure 4(a) is incorrect because $p_0$, the type of $r_1$ in the second instruction cannot be inferred. Notice that these annotations can be inferred from the sMIRA/sFORD programs; they are not present in the final LTL code.

### 3.1 Operational Semantics of sFORD Programs

We define an abstract machine to evaluate sFORD programs. The state $M$ of this machine is defined in terms of a tuple with four elements: $(C, D, R, I)$. If $M$ is a program state and we have $M'$ such that $M \to M'$, then we say that $M$ can *take a step.* A program state $M$ is *stuck* if $M$ cannot take a step. A program state $M$ *goes wrong* if $\exists M' : M \to^* M'$ and $M'$ is stuck. $I$ is defined in Figure 3(a); the code heap $C$, data heap $D$, and register bank $R$ are defined below.

$$\begin{array}{ll} \text{(Code Heap)} & C ::= \{L_1 = I_1, \ldots, L_k = I_k\} \\ \text{(Data Heap)} & D ::= \{l_1 = p_1, \ldots, l_m = p_m\} \\ \text{(Register Bank)} & R ::= \{r_1 = p_1, \ldots, r_n = p_n\} \\ \text{(Machine State)} & M ::= (C, D, R, I) \end{array}$$

The evaluation rules for our abstract machine are given in Figure 5. Rules 1, 2 and 3 evaluate the operands of sFORD. The assignment statement (Rule 4) modifies the mapping in the register bank, and the store statement (Rule 5) modifies the mapping in the data heap. The result of a conditional branch has no importance in our representation. Therefore, an instruction such as if $(r, p)$ jump $v$ is evaluated non-deterministically by either Rule 6 or Rule 7. We conservatively assume that a call instruction changes the contents of all the caller save registers. It also defines a register with the return value (Rule 8). In order to simulate the effects of a function call on the caller-save registers we define the erasing function "$\diamond$" below. We augment the set of pseudo-registers with $\perp$. This pseudo will be used as the type of non-initialized registers and we assume that it is not defined in any instruction of the original sMIRA program.

$$\begin{array}{l} \diamond : R \times X \mapsto R' \\ (R \diamond X)(r) = \perp \ \text{if} \ r \in X, \ \text{else} \ R(r) \end{array}$$

In Figure 5 the set $X$ is replaced by the set of caller-save registers. We draw the attention of the reader to the premises of rules 4 to 8, which ensure that a location used by an instruction indeed contains the pseudo that is expected by that instruction. For example, for the sFORD instruction $(r_1, p_1) = (r_0, p_0)$, the premise of Rule 4 ensures that $r_0$ is holding the value $p_0$ when this instruction is executed.

$$D, R \vdash \bullet \tag{1}$$

$$D, R \vdash (r, p), \mathsf{if}\ \ R(r) = p \wedge p \neq \bot \tag{2}$$

$$D, R \vdash (l, p), \mathsf{if}\ \ D(l) = p \wedge p \neq \bot \tag{3}$$

$$\frac{D, R \vdash o}{(C, D, R, (r, p) = o; I) \rightarrow (C, D, R[r \mapsto p], I)} \tag{4}$$

$$\frac{D, R \vdash o}{(C, D, R, (l, p) = o; I) \rightarrow (C, D[l \mapsto p], R, I)} \tag{5}$$

$$\frac{D, R \vdash (r, p)}{(C, D, R, \mathsf{if}\ \ (r, p)\ \mathsf{jump}\ \ L; I) \rightarrow (C, D, R, I')}\ \ C_{cond} \tag{6}$$

$$C_{cond} = L \in \mathsf{domain}(C) \wedge C(L) = I'$$

$$\frac{D, R \vdash (r, p)}{(C, D, R, \mathsf{if}\ \ (r, p)\ \mathsf{jump}\ \ L; I) \rightarrow (C, D, R, I)} \tag{7}$$

$$\frac{\forall (r_i, p_i), 1 \leq i \leq s, D, R \vdash (r_i, p_i)}{\begin{array}{c}(C, D, R, (r_0, p_0) = \mathsf{call}\ \ (r_1, p_1), \ldots, (r_s, p_s); I) \\ \rightarrow (C, D, (R \diamond \mathsf{callerSave})[r_0 \mapsto p_0], I)\end{array}} \tag{8}$$

$$(C, D, R, \mathsf{jump}\ \ L) \rightarrow (C, D, R, I)\ \ \mathsf{if}\ \ C_{jump} \tag{9}$$

$$C_{jump} = L \in \mathsf{domain}(C) \wedge C(L) = I$$

**Fig. 5.** Operational Semantics of sFORD programs

### 3.2 Typing Rules

We define the following types for values:

$$\text{value types } t ::= p \ \mid\ \mathsf{Const}$$

We define three typing environments:

$$\begin{array}{ll}
\text{(Code heap type)} & \Psi ::= \{L_1 : (\Gamma_1 \times \Delta_1), \\
 & \quad\quad \ldots, L_k : (\Gamma_k \times \Delta_k)\} \\
\text{(Register bank type)} & \Gamma ::= \{r_1 : p_1, \ldots, r_m : p_m\} \\
\text{(Data heap type)} & \Delta ::= \{l_1 : p_1, \ldots, l_n : p_n\}
\end{array}$$

Operands that have no effect on the register allocation process are given the type Const. The type of a machine location (register or memory address) is determined

Operands

$$\vdash \bullet : \mathsf{Const} \tag{10}$$

$$\frac{\mathit{\Gamma}(r) = p \qquad p \neq \bot}{\mathit{\Gamma} \vdash (r, p) : p} \tag{11}$$

$$\frac{\mathit{\Delta}(l) = p \qquad p \neq \bot}{\mathit{\Delta} \vdash (l, p) : p} \tag{12}$$

Instructions

$$\frac{\mathit{\Delta}; \mathit{\Gamma} \vdash o : t \qquad p \neq \bot}{\mathit{\Psi} \vdash (r, p) = o : (\mathit{\Gamma} \times \mathit{\Delta}) \mapsto (\mathit{\Gamma}[r : p] \times \mathit{\Delta})} \tag{13}$$

$$\frac{\mathit{\Delta}; \mathit{\Gamma} \vdash o : t \qquad p \neq \bot}{\mathit{\Psi} \vdash (l, p) = o : (\mathit{\Gamma} \times \mathit{\Delta}) \mapsto (\mathit{\Gamma} \times \mathit{\Delta}[l : p])} \tag{14}$$

$$\frac{\mathit{\Gamma} \vdash (r, p) : p \qquad \mathit{\Psi} \vdash L : (\mathit{\Gamma}' \times \mathit{\Delta}') \qquad (\mathit{\Gamma} \times \mathit{\Delta}) \leq (\mathit{\Gamma}' \times \mathit{\Delta}')}{\mathit{\Psi} \vdash \mathsf{if}\ (r, p)\ \mathsf{jump}\ L : (\mathit{\Gamma} \times \mathit{\Delta}) \mapsto (\mathit{\Gamma} \times \mathit{\Delta})} \tag{15}$$

$$\frac{\forall (r_i, p_i), 1 \leq i \leq s, \mathit{\Gamma} \vdash (r_i, p_i) : p_i \qquad p_0 \neq \bot}{\mathit{\Psi} \vdash (r_0, p_0) = \mathsf{call}\ (r_1, p_1), \ldots, (r_s, p_s)\ \ : (\mathit{\Gamma} \times \mathit{\Delta}) \mapsto ((\mathit{\Gamma} \diamond \mathsf{callerSave})[r_0 : p_0] \times \mathit{\Delta})} \tag{16}$$

Instruction sequences

$$\frac{\mathit{\Psi} \vdash L : (\mathit{\Gamma}' \times \mathit{\Delta}') \qquad (\mathit{\Gamma} \times \mathit{\Delta}) \leq (\mathit{\Gamma}' \times \mathit{\Delta}')}{\mathit{\Psi} \vdash \mathsf{jump}\ L : (\mathit{\Gamma} \times \mathit{\Delta})} \tag{17}$$

$$\frac{\mathit{\Psi} \vdash i : (\mathit{\Gamma} \times \mathit{\Delta}) \mapsto (\mathit{\Gamma}' \times \mathit{\Delta}') \qquad \mathit{\Psi} \vdash I : (\mathit{\Gamma}' \times \mathit{\Delta}')}{\mathit{\Psi} \vdash i; I : (\mathit{\Gamma} \times \mathit{\Delta})} \tag{18}$$

Bank of Registers

$$\frac{\forall r \in \mathsf{domain}(\mathit{\Gamma}).\ \vdash R(r) : \mathit{\Gamma}(r)}{\mathit{\Psi} \vdash R : \mathit{\Gamma}} \tag{19}$$

Data Heap

$$\frac{\forall l \in \mathsf{domain}(\mathit{\Delta}).\ \vdash D(l) : \mathit{\Delta}(l)}{\mathit{\Psi} \vdash D : \mathit{\Delta}} \tag{20}$$

Code Heap

$$\frac{\forall L \in \mathsf{domain}(\mathit{\Psi}).\mathit{\Psi} \vdash C(L) : \mathit{\Psi}(L)}{\vdash C : \mathit{\Psi}} \tag{21}$$

Machine states:

$$\frac{\vdash C : \mathit{\Psi} \qquad \vdash D : \mathit{\Delta} \qquad \vdash R : \mathit{\Gamma} \qquad \mathit{\Psi} \vdash I : (\mathit{\Gamma}' \times \mathit{\Delta}') \qquad (\mathit{\Gamma} \times \mathit{\Delta}) \leq (\mathit{\Gamma}' \times \mathit{\Delta}')}{\vdash (C, D, R, I)} \tag{22}$$

**Fig. 6.** Type System of sFORD

by the pseudo that is stored in that location. The environment $\varGamma$ contains the types of the machine registers, and the typing environment $\varDelta$ contains the types of locations in the data heap. We will refer to $\varGamma$ and $\varDelta$ as *location environments*. The environment $\varPsi$ determines the type of each instruction block. The type of an instruction sequence is given by the minimum configuration of the bank of registers and data heap that the sequence must receive in order to be able to execute properly. We consider instructions as functions that modify the location environments, that is, an instruction $i$ expects an environment $(\varGamma \times \varDelta)$ and returns a possibly modified environment $(\varGamma' \times \varDelta')$. We define an ordering on location environments as follows:

$$\varGamma \leq \varGamma' \ \ \mathsf{if}\ \forall r,\ \ r : p \in \varGamma' \ \mathsf{then}\ r : p \in \varGamma \tag{23}$$

$$\varDelta \leq \varDelta' \ \ \mathsf{if}\ \forall l,\ \ l : p \in \varDelta' \ \mathsf{then}\ l : p \in \varDelta \tag{24}$$

$$(\varGamma \times \varDelta) \leq (\varGamma' \times \varDelta') \ \ \mathsf{if}\ \varGamma \leq \varGamma' \wedge \varDelta \leq \varDelta' \tag{25}$$

The type rules for sFORD programs are given in Fig. 6. According to rule 11, the type of a register binding such as $(r, p)$ is the temporary $p$, but, only if $p$ is the type of $r$ in the $\varGamma$ environment, otherwise it does not type-check. Similarly, Rule 12 determines the type of memory bindings. Rules 13, 14 and 16 change the location environment. The ordering comparison in the premises of Rules 15 and 17 is necessary to guarantee that all the registers alive at the beginning of an instruction block have well defined types.

None of the programs in Fig. 4 type-check. In Fig. 4(a), $r_1$ is not declared with type $p_0$, thus the premise of Rule 11 is not satisfied. In Fig. 4(b), the type of $r_0$, before the execution of instruction 3, is $p_1$, not $p_0$, as expected by the type annotation. Again, Rule 11 is not satisfied. Fig. 4(c) presents a similar case, but using a memory location instead of a register: the type of $l_0$ in instruction 3 is not $p_0$, as expected, but $p_1$. The type of the used operand would not satisfy the premise in Rule 12. In Fig. 4(d), $r_1$ has type $\perp$ before the execution of instruction 3, which is different from the expected type $p_1$. Finally, in Fig. 4(e) $\varPsi(L_2) = ([r_0 : p_0] \times \varDelta)$, but the type of instruction 4 is $([r_0 : p_1] \times \varDelta) \mapsto ([r_0 : p_1] \times \varDelta)$. This would not satisfy the inequality in Rule 17.

### 3.3 Type Soundness

We state the lemmas and theorems that constitute our soundness proof. Our soundness proof assumes that the sMIRA program defines each pseudo $p$ before $p$ is used. If the sFORD program type-checks, then it preserves the values alive in the original sMIRA code.

**Theorem 1 (Preservation).** *If* $\vdash M$*, and* $M \rightarrow M'$*, then* $\vdash M'$*.*

**Lemma 1 (Canonical Values).** *If* $\vdash C : \Psi$, $\vdash D : \Delta$ *and* $\vdash R : \Gamma$ *then:*

1. *If* $\Psi \vdash L : (\Gamma \times \Delta)$, *then* $L \in \mathsf{domain}(C)$, $C(L) = I$ *and* $\Psi \vdash I : (\Gamma \times \Delta)$.
2. *If* $\Delta; \Gamma \vdash o : p$, *then* $o = r$, *or* $o = l$. *If* $o = r$, *then* $r \in \mathsf{domain}(R)$, *else if* $o = l$, *then* $l \in \mathsf{domain}(D)$.
3. *If* $\Delta; \Gamma \vdash o : \mathsf{Const}$, *then* $o = \bullet$.

**Theorem 2 (Progress).** *If* $\vdash M$, *then* $M$ *is a final state, or there exists* $M'$ *such that* $M \mapsto M'$.

**Corollary 1 (Soundness).** *If* $\vdash M$, *then* $M$ *cannot go wrong.*

We have checked the proof using Twelf [32], and this proof can be found at `http://compilers.cs.ucla.edu/ralf/twelf/`

### 3.4 Preservation of Callee-Save Registers

Our type system is intra-procedural, and it assumes that a function call preserves callee-save registers. This must be verified for each procedure, after its type-checking phase, when every instruction has a well know type. If we assume that a machine register is either caller-save or callee-save, this verification step can be done via a simple test. Let $L_0$ be the label of the first instruction in the procedure, and let $L_e$ be an exit point. Let $\Psi(L_0) = (\Gamma_0 \times \Delta_0)$, and let $\Psi(L_e) = (\Gamma_e \times \Delta_e)$. Callee-save registers are preserved at exit point $L_e$, if $\Gamma_e \diamond \mathsf{CallerSave} \leq \Gamma_0 \diamond \mathsf{CallerSave}$.

Along with the preservation of callee-save registers, our type systems has a set of consistency requirements, which can be carried out as a sequence of table lookup verifications. These checks are explained in the full version of this paper, available at `http://compiler.cs.ucla.edu/ralf`.

## 4 Experimental Results

Figure 1 presents a high-level block diagram of RALF. RALF interfaces the transformation between the RTL and LTL intermediate representations used by gcc. RALF's front end consists mainly of gcc's parser, gcc's optimization phases, and code to produce a MIRA program from a RTL program. The back end consists mostly of a type checker, code for producing LTL instructions, and gcc's code generation engine. RALF interacts with the implementation of a register allocator via ordinary ASCII files. Given a RTL program $P$, RALF translates $P$ into a MIRA ASCII program $\mathcal{M}$, which is then fed to the plugged-in register allocator. The register allocator outputs a set of FORD directives $\mathcal{F}$, which are then given back to RALF. RALF checks that $(\mathcal{M}, \mathcal{F})$ is a correct mapping via the type system described in Section 3, applies the directives $\mathcal{F}$ on the original RTL program, and generates gcc's LTL code. RALF's back end does not need the original MIRA file in order to produce the LTL program. When producing the RTL code, RALF inserts loads and stores for callee save registers at the entrance and exit of each function (a smart register allocator might decide to take on that responsibility itself and RALF has an option for that).

We have tested RALF with eight different register allocators: (1: gcc -O2) the allocator present in the gcc compiler, which has two main phases: (a) aggressive register allocation for local variables within basic blocks, (b) conservative allocation for the whole function. (2: Naive) The *naive* register allocator, which spills all the pseudos. (3: UBC) usage count based register allocator [11], (4: IRC) iterated register coalescing [13], (5: Chordal) register allocation via coloring of chordal graphs [27], (6: LS) linear scan [28], (7: RA) integer linear program (ILP) [23], (8: SARA) stack location allocation combined with register allocation (SARA) [23].

Five of the register allocators have been implemented in Java (2, 3, 4, 5 and 6). Algorithms 7 and 8 have been implemented in AMPL [10]. The interface provided by RALF is extremely simple, and most of the code used to parse and output MIRA/FORD files could be reused among the different implementations. Table 7 compares the size of each implementation; (J) stands for Java, and (A) for AMPL. We do not compare the execution speed of the allocators, because they have been implemented in different languages.

| RA | #LOC | |
|---|---|---|
| | RA | Interface |
| Naive | 48 (J) | 773 (J) |
| UBC | 2766 (J) | 773 (J) |
| IRC | 3538 (J) | 773 (J) |
| Chordal | 4134 (J) | 773 (J) |
| LS | 385 (J) | 1100 (J) |
| RA | 495 (A) | 298 (A) |
| SARA | 731 (A) | 400 (A) |

**Fig. 7.** Comparison between different register allocators plugged on RALF

We have plugged each of the eight algorithms into RALF and then tested the produced code on a StrongARM/XScale processor, with 64MB SDRAM, and no cache. We have drawn our benchmark programs from a variety of sources. We chose these benchmarks in part because the more traditional ones (for example, SPEC) have a huge memory print, and cannot be run in our resource constrained ARM hardware.

- Stanford Benchmark suite: a collection of seven programs that test recursive calls and array indexing.
- NetBench [19]: url is a network related benchmark that implements HTTP based switching; md5 is a typical cryptographic algorithm.
- Pointer-intensive benchmark [7]: This benchmark suite is a collection of pointer-intensive benchmarks. Yacr2 is an implementation of a maze solver and Ft is an implementation of a minimum spanning tree algorithm.

| | LOC | RTL | gcc-O2 | | LS | | RA | | SARA | | Chordal | | IRC | | UCB | | Naive | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bench | | | mem | csr | mem | csr | mem | csr | mem | csr | mem | csr | mem | csr | mem | csr | mem | csr |
| Stanford | 307 | 1082 | 20 | 81 | 171 | 107 | 22 | 63 | 24 | 69 | 34 | 134 | 70 | 100 | 44 | 133 | 854 | 0 |
| yacr2 | 3979 | 10838 | 1078 | 289 | 3035 | 335 | 1003 | 123 | 1109 | 142 | 2121 | 357 | 1957 | 314 | 2200 | 361 | 8181 | 0 |
| ft | 2155 | 3218 | 299 | 130 | 538 | 151 | 225 | 87 | 230 | 106 | 371 | 162 | 561 | 100 | 360 | 169 | 2184 | 0 |
| c4 | 897 | 40948 | 187 | 123 | 715 | 301 | 176 | 145 | 179 | 151 | 416 | 170 | 453 | 145 | 394 | 170 | 3531 | 0 |
| mm | 885 | 3388 | 386 | 92 | 2494 | 68 | 375 | 116 | 380 | 92 | 591 | 93 | 648 | 93 | 687 | 93 | 2590 | 0 |
| url | 652 | 1264 | 102 | 54 | 313 | 16 | 120 | 56 | 120 | 58 | 155 | 61 | 201 | 51 | 203 | 63 | 860 | 0 |
| md5 | 790 | 3464 | 519 | 110 | 1869 | 433 | 500 | 120 | 500 | 120 | 570 | 228 | 697 | 174 | 580 | 229 | 2714 | 0 |

**Fig. 8.** Compile time statistics

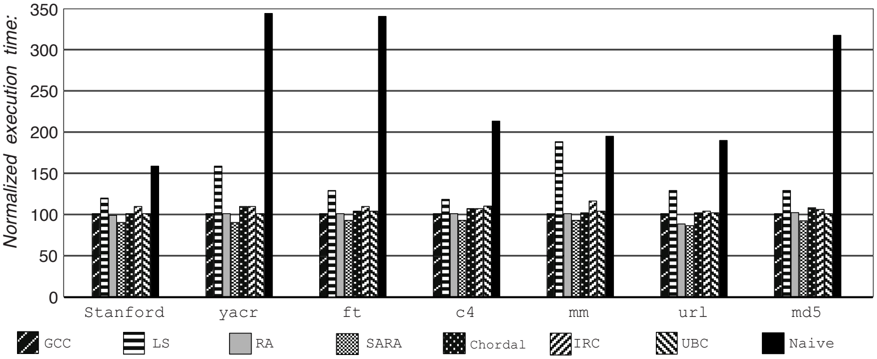


**Fig. 9.** Comparison of different register allocators using execution time of benchmarks as the metric

- c4 and mm are taken from the comp.benchmarks USENET newsgroup at http://www.cs.wisc.edu/~thomas/comp.benchmarks.FAQ.html. c4 is an implementation of the connect-4 game, and mm is a matrix multiplication benchmark.

The number of lines of C code (LOC) and the number of instructions in the RTL for these benchmarks are presented in Figure 8. These benchmarks are non-floating point programs. (We had to edit few of the programs to remove some code that uses floating point operations; we did so only after ensuring that the code with floating point operations is not critical to the behavior of the program.) For each benchmark, Figure 8 presents two static compile time statistics: the number of memory accesses (mem) due to spill/reload instructions, and the number of callee save registers (csr) used by the register allocator (leads to more memory accesses).

The chart in Figure 9 compares the execution times of the programs produced by each of the register allocators. The execution times have been normalized against the time obtained by programs compiled with gcc at the -O2 optimization level. It can be noted that, although the gcc algorithm is heavily tuned for the StrongARM architecture, Chordal, UBC, and IRC present comparative performances. Also Chordal and IRC's performances are similar, which confirms the results found by Pereira et al [27]. Figure 9 suggests an upper limit on the gains that any register allocator can make. Even in the most extreme case, the code generated by the naive allocator is worse by a factor of 2.5 (as compared to the optimal solution found by the ILP based allocator). An important point is that most of these benchmarks deal with structures and arrays that require compulsory memory accesses, and it seems that these accesses overshadow the spill cost and hence such a small (2.5 times) improvement. Another conclusion is that the obtained execution time given by optimal solutions (SARA and RA), that run in worst-case exponential time, is not much lower than that obtained by polynomial time heuristics. It should be pointed that our experiments compare specific implementations of the allocators, and are run on a specific target machine; however, it was our objective to be as faithful as possible to the original description of each algorithm.

## 5 Conclusion

We have presented a framework that facilitates the development of register allocation algorithms. Our framework consists of the two description languages MIRA and FORD, plus a type system. Our framework is easy to use, and versatile enough to support a wide variety of register allocation paradigms. In order to validate this claim, we have developed RALF, an implementation of our framework for the StrongARM architecture, and used it to compare eight different register allocators.

Our framework has several limitations which may be overcome in future work: FORD does not permit the register allocator to modify the control flow graph of the target program; the grammar of MIRA supports only intra-procedural register allocation; the validation algorithm does not handle bitwidth aware register allocation [29]; and FORD does not support the concepts of re-materialization or code-motion. The implementation of RALF itself has the limitations that it targets only the ARM architecture, and RALF does not handle pseudos of type float or double (we have opted for this restriction to keep the implementation simple). So far we have experimented with only the C front-end of gcc; RALF can be seamlessly used with any language supported by gcc. Currently, we are extending RALF to allow the register allocator to do bitwidth-sensitive-analysis.

RALF, our benchmarks, our Twelf proof, and a collection of tools that we have developed to aid in the design and test of register allocators are publicly available at `http://compilers.cs.ucla.edu/ralf`.

**Acknowledgments.** We thank the anonymous reviewers for comments on a draft of the paper. We were supported in part by a National Science Foundation Award number 0401691. Fernando Pereira was sponsored by the Brazilian Ministry of Education under grant number 218603-9.

## References

1. Standard ML of New Jersey (2000), `http://www.smlnj.org/`
2. GNU C compiler (2005), `http://gcc.gnu.org`
3. Adl-Tabatabai, A.-R., Gross, T., Lueh, G.-Y.: Code reuse in an optimizing compiler. In: OOPSLA, pp. 51–68. ACM Press, New York (1996)
4. Agat, J.: Types for register allocation. In: Clack, C., Hammond, K., Davie, T. (eds.) IFL 1997. LNCS, vol. 1467, pp. 92–111. Springer, Heidelberg (1998)
5. Andersson, C.: Register allocation by optimal graph coloring. In: Hedin, G. (ed.) CC 2003 and ETAPS 2003. LNCS, vol. 2622, pp. 34–45. Springer, Heidelberg (2003)
6. Appel, A.W, George, L.: Optimal spilling for CISC machines with few registers. In: PLDI, pp. 243–253. ACM Press, New York (2001)
7. Austin, T.M., Breach, S.E., Sohi, G.S.: Efficient detection of all pointer and array access errors. In: PLDI, pp. 290–301 (1994)
8. Chaitin, G.J.: Register allocation and spilling via graph coloring. SIGPLAN Notices 17(6), 98–105 (1982)
9. Elleithy, K.M., Abd-El-Fattah, E.G.: A genetic algorithm for register allocation. In: Ninth Great Lakes Symposium on VLSI, pp. 226–227 (1999)
10. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL A modeling language for mathematical programming. Scientific Press (1993), `http://www.ampl.com`
11. Freiburghouse, R.A.: Register allocation via usage counts. Commun. ACM 17(11), 638–642 (1974)
12. Fu, C., Wilken, K.: A faster optimal register allocator. In: MICRO, pp. 245–256. IEEE Computer Society Press, Los Alamitos (2002)
13. George, L., Appel, A.W.: Iterated register coalescing. TOPLAS 18(3), 300–324 (1996)
14. Goodwin, D.W., Wilken, K.D.: Optimal and near-optimal global register allocations using 0-1 integer programming. SPE 26(8), 929–968 (1996)
15. Hall, M.W., Anderson, J.-A.M., Amarasinghe, S.P., Murphy, B.R., Liao, S.-W., Bugnion, E., Lam, M.S.: Maximizing multiprocessor performance with the SUIF compiler. IEEE Computer 29(12), 84–89 (1996)
16. Huang, Y., Childers, B.R., Soffa, M.L.: Catching and identifying bugs in register allocation. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, Springer, Heidelberg (2006)
17. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL, pp. 42–54. ACM Press, New York (2006)
18. Lueh, G.-Y., Gross, T., Adl-Tabatabai, A.-R.: Global register allocation based on graph fusion. In: Languages and Compilers for Parallel Computing, pp. 246–265 (1996)
19. Memik, G., Mangione-Smith, B., Hu, W.: Netbench: A benchmarking suite for network processors. In: IEEE International Conference Computer-Aided Deisgn, IEEE Computer Society Press, Los Alamitos (2001)
20. Morrisett, G., Walker, D., Crary, K., Glew, N.: From system F to typed assembly language. TOPLAS 21(3), 527–568 (1999)
21. Naik, M., Palsberg, J.: Compiling with code size constraints. Transactions on Embedded Computing Systems 3(1), 163–181 (2004)
22. Krishna Nandivada, V., Palsberg, J.: Efficient spill code for SDRAM. In: CASES, pp. 24–31 (2003)
23. Krishna Nandivada, V., Palsberg, J.: Sara: Combining stack allocation and register allocation. In: Bodik, R. (ed.) CC 2005. LNCS, vol. 3443, pp. 232–246. Springer, Heidelberg (2005)

24. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI, pp. 83–95. ACM Press, New York (2000)
25. Necula, G.C., Lee, P.: The design and implementation of a certifying compiler. In: PLDI, pp. 333–344 (1998)
26. Ohori, A.: Register allocation by proof transformation. Science of Computer Programming 50(1-3), 161–187 (2004)
27. Pereira, F.M.Q., Palsberg, J.: Register allocation via coloring of chordal graphs. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, Springer, Heidelberg (2005)
28. Poletto, M., Sarkar, V.: Linear scan register allocation. TOPLAS 21(5), 895–913 (1999)
29. Tallam, S., Gupta, R.: Bitwidth aware global register allocation. In: POPL, pp. 85–96 (2003)
30. Traub, O., Holloway, G.H., Smith, M.D.: Quality and speed in linear-scan register allocation. In: PLDI, pp. 142–151 (1998)
31. Vallee-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: CASCON (1999)
32. Pfenning, F., Schürmann, C.: System description: Twelf - a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) Automated Deduction - CADE-16. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)

# A New Algorithm for Identifying Loops in Decompilation*

Tao Wei, Jian Mao, Wei Zou**, and Yu Chen

Institute of Computer Science and Technology
Peking University
{weitao,maojian,zouwei,chenyu}@icst.pku.edu.cn

**Abstract.** Loop identification is an essential step of control flow analysis in decompilation. The Classical algorithm for identifying loops is Tarjan's interval-finding algorithm, which is restricted to reducible graphs. Havlak presents one extension of Tarjan's algorithm to deal with irreducible graphs, which constructs a loop-nesting forest for an arbitrary flow graph. There's evidence showing that the running time of this algorithm is quadratic in the worst-case, and not almost linear as claimed. Ramalingam presents an improved algorithm with low time complexity on arbitrary graphs, but it performs not quite well on "real" control flow graphs (CFG). We present a novel algorithm for identifying loops in arbitrary CFGs. Based on a more detailed exploration on properties of loops and depth-first search (DFS), this algorithm traverses a CFG only once based on DFS and collects all information needed on the fly. It runs in approximately linear time and does not use any complicated data structures such as Interval/Derived Sequence of Graphs (DSG) or UNION-FIND sets. To perform complexity analysis of the algorithm, we introduce a new concept called *unstructuredness coefficient* to describe the unstructuredness of CFGs, and we find that the unstructuredness coefficients of these executables are usually small (<1.5). Such "low-unstructuredness" property distinguishes these CFGs from general single-root connected directed graphs, and it offers an explanation why those algorithms existed perform not quite well on real-world cases. The new algorithm has been applied to 11526 CFGs in 6 typical binary executables on both Linux and Window platforms. Experimental result has validated our theoretical analysis and it shows that our algorithm runs 2-5 times faster than the Havlak-Tarjan algorithm, and 2-8 times faster than the Ramalingam-Havlak-Tarjan algorithm.

**Keywords:** Control flow analysis, Decompilation, Loop identifying, Unstructuredness coefficient.

## 1 Introduction

Decompilation is a key technique for static analysis in the field of reverse engineering. Decompilation was initially introduced for porting programs across

* Supported by The National High Technology Research and Development Program of China (No. 2006AA01Z402).

** Corresponding author.

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 170–183, 2007.

platforms. It then had been widely used in areas such as software maintenance, re-engineering and comprehension of legacy systems. Since the 1990s, demand on decompilation from software security analysis community has been growing rapidly due to outbreaks of security vulnerabilities and malicious codes[1].

When decompiling a program, it is important to analyze its control flow to correctly recover the underlying structures, such as loops, 2-way branches and n-way branches, from its corresponding binary executable. This paper mainly focuses on how to identify loops.

Loops are control structures used for repeating instructions. In control flow graphs (CFG) [2], loops are often nested within other loops. Such phenomenon induces a structure called "loop-nesting forest" [3][4]. Furthermore, although structured programming is well adopted by modern programmers, irreducible loops (loops with multientry) [5][6] still widely exist in executable codes due to optimizations performed by compilers. Identifying loops, in particular nested and irreducible ones, is a major challenge for the task of decompilation.

In 1970 F.E. Allen and J. Cocke pioneered the work on identifying loops by introducing the concept of *reducibility*[5][6] for control flow graphs. Since then both compiler and decompiler research communities have been investigating this problem. Influential pieces of work include those done by R.E. Tarjan [8], P. Havlak [7] and G. Ramalingam[3]. However, loop identification schemes proposed in these work are often based on multi-pass traversals and complicated data structures, such as *Interval/Derived sequence of graphs (DSG)*[5][6] and *UNION-FIND sets*[9]; these data structures often require complex operations, and such operations slow down the loop identification schemes [10].

This paper presents an innovative algorithm for identifying loops in binary executables. We explore some useful properties of loops and depth-first search (DFS) which make DFS collecting more information than simple forward/cross/backward edge information. Based on these properties, we give an algorithm which uses a one-pass DFS traversal to solve all loop problems. This algorithm does not use any complicated data structures, so it is simple and easy to implement.

We have applied our algorithm and other classic algorithms to 11526 CFGs in 6 typical binary executables on Windows XP and Linux. The experiments show that our algorithm runs 2-5 times faster than the Havlak-Tarjan algorithm [7], and 2-8 times faster than the Ramalingam-Havlak-Tarjan algorithm [3].

Furthermore, as an interesting byproduct of complexity analysis of this algorithm, we introduce a new concept called ***unstructuredness coefficient***. This coefficient could describe the unstructuredness of CFG.

The statistics of experiments shows that while most real-world binary executables have irreducible CFGs, unstructuredness coefficients of CFGs are usually smaller than 1.5 and hardly correlated to the size of CFG.

Such "low-unstructuredness" property distinguishes these CFGs from general single-root connected directed graphs, and it offers an explanation of why those algorithms with low time complexity for arbitrary graphs perform not quite well on "real" CFGs, esp. the Ramalingam-Havlak-Tarjan algorithm.

Besides decompilation, our algorithm could be used in many applications, such as computing the iterated dominance frontier for the SSA form and Sparse Evaluation

Graphs, constructing the dominator tree[4], and sequentializing program dependence graphs for code generation[12].

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 provides terminology and notations of identifying loops. Section 4 presents the algorithm for identifying loops. Section 5 introduces the concept of unstructuredness coefficient, and presents the complexity analysis of our scheme. Section 6 reports our experimental result and finding, in particular, the statistics of unstructured coefficient in real-world binary executables. Section 7 concludes this paper.

## 2 Related Work

Identifying loops is a well-built problem in control-flow analysis area. Loops in CFG have more attributes than simple cycles, such as nesting, multi-entry and irreducibility. Hence identifying loops in CFG is generally more challenging than detecting cycles. Research on identifying loops has a long history, starting from 1970 when F.E. Allen and J. Cocke introduced the concept of reducibility[5][6] for control flow graphs. Since then many researchers in both compiler and decompiler fields have studied this problem extensively.

Reducibility is an important property of CFG on its structuredness. In 1972 M.S.Hecht and J.D.Ullman showed that all and only the irreducible CFGs have a multi-entry-loop subgraph [11], as shown in Fig.1.

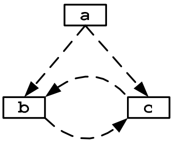


**Fig. 1.** The irreducible core

In CFG, loops are often nested within other loops according to their headers' positions. Such phenomenon induces a structure called "loop-nesting forest" [3][4]. Fig.2(A) shows a CFG with nested loops, and Fig.2(B) shows the corresponding loop-nesting forest.

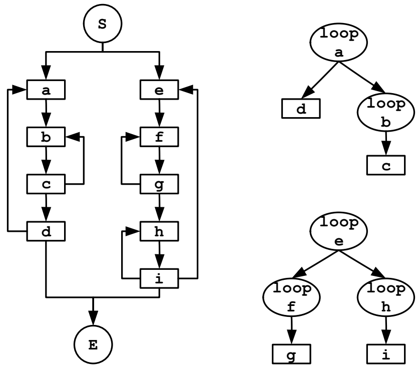
There are various definitions of loop and loop-nesting forest. While there is a well-accepted one by Tarjan [8] of loops in a reducible graph, there is no consensus on how the loop nesting forest should be defined for CFGs with nested loops. B. Steensgaard [12], V.C. Sreedhar *et al* [13], Havlak [7] and Ramalingam [4] each provided a different definition.

Consider the CFG shown in Fig.3(A): The Sreedhar–Gao–Lee algorithm [13] and the Ramalingam algorithm [4] both identify a single loop ***{a,b,c,d}***; the Steensgaard algorithm [12] identifies two loops ***{a,b,c,d}*** and ***{b,c}***; the Havlak algorithm [7] identifies three loops ***{a,b,c,d}***, ***{b,c,d}*** and ***{c,d}***, as shown in Fig.3 (B).

In the context of decompilation, only the definition given by Havlak meets the requirements for rebuilding high level structures using single-entry loops and minimum ***goto*** (for re-entry edges) statements. Hence in this paper we adopt his definition.

Under Havlak's definition, the classical algorithm for identifying loops is Tarjan's interval-finding algorithm [8] proposed in 1974, which is restricted to reducible graphs. In 1997 Havlak presented an extension [7] to Tarjan's algorithm, which can handle arbitrary flow graphs.

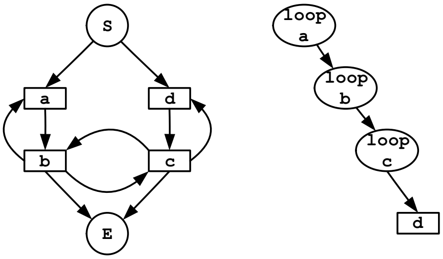
The Havlak-Tarjan algorithm traverses a CFG twice: first a top-down traversal based on depth-first search, which collects information of forward edges, cross edges and back edges; then a bottom-up traverse based on the UNION-FIND operation, which propagates loop header information backward from loop tails.



(A) a CFG with nested loops

(B) the loop-nesting forest of A

**Fig. 2.** A CFG with nested loops and its loop-nesting forest: loop ***b*** is nested in loop ***a***; loop ***f*** and loop ***h*** are nested in loop ***e***



(A) an irreducible CFG

(B) the loop-nesting forest of A

**Fig. 3.** A classic irreducible CFG and its loop-nesting forest based on the definition by Havlak

In 1999 Ramalingam showed that the running time of the Havlak algorithm is quadratic when the target CFG's multientry unstructuredness is very high; he then modified the algorithm to make it run in almost linear time [3]. However, the Ramalingam-Havlak-Tarjan algorithm needs extra procedures to solve least common ancestors and to mark irreducible loops, and these procedures need UNION-FIND operations.

On the other hand, the Steensgaard algorithm runs in quadratic time, and the Sreedhar-Gao-Lee algorithm runs in almost linear time, but the latter requires the dominator tree being built in advance.

In 2001, K.D. Cooper showed that, based on empirical evidence complex operations required by UNION-FIND slow down programs in practice; furthermore, simple algorithms with discouraging asymptotic complexities might be faster in handling real-world cases than those running in almost linear time, but containing complex operations [10].

# 3 Preliminaries

This section briefly describes some basic concepts in control flow analysis and definitions about loops, and introduces some important properties of loop and DFS.

## 3.1 Concepts in Control Flow Analysis

We present brief descriptions of concepts in control flow analysis as following. The detailed version can be found in [2] and [9].

The instructions of a program are organized into ***basic blocks***, where program flow enters a basic block at its first instruction and leaves the basic block at its last instruction.

A ***control flow graph (CFG)*** is a single-root, connected and directed graph for describing control flow information of a program□ it is often represented by a triple $(N,E,h)$, where $N$ is the set of basic blocks of the underlying program, $E$ is the set of directed edges between these basic blocks, and $h$ is the entry of the program.

For a basic block $b$, $Succ(b)$ is the set of successors of $b$, and $Pred(b)$ is the set of predecessors of $b$.

A path from a node $u$ to a node $u'$ in a graph $G=(N,E,h)$ is a sequence $<v_0,v_1,v_2,...,v_k>$ of nodes such that $u=v_0$, $u'=v_k$, and $<v_{i-1},v_i> \in E$ for $i=1,2,...,k$.

A ***depth-first search (DFS)*** of a CFG $G=(N,E,h)$ visits all the nodes, marking them after they have been visited. The next node visited is an unmarked successor of the most recently visited node with such a successor. The time complexity of DFS is $O(N+E)$.

If a DFS traversal begins with $h$, and all other nodes are reachable, the edges followed define a ***depth-first spanning tree (DFST)*** of $G$.

$DFSP(N)$, the ***depth-first search path*** of node $N$, is the path from $h$ to $N$ in the DFST of $G$. Given a node $N$ and a node $M$, and $N$ is in $DFSP(M)$, then $DFSP(N,M)$ is the part of $DFSP(M)$ from $N$ to $M$.

Besides creating a depth-first spanning tree, depth-first search also timestamps each node. Each node $v$ has two timestamps: the ***first timestamp*** $d[v]$ records when $v$ is first discovered, and the ***second timestamp*** $f[v]$ records when the search finishes examining $v$'s adjacency list. ***Parenthesis theorem*** is an important property of DFS, and here is a short description: for two node $u$ and $v$, the two sets $[d[u],f[u]]$ and $[d[v],f[v]]$ are either disjoint or nested.

In this paper, we use three edge types in terms of the DFST $G_T$ produced by a DFS on $G$:

- ***Back edges*** are those edges $<u,v>$ connecting a vertex $u$ to an ancestor $v$ in $G_T$. Self-loops are considered to be back edges. An edge $<u,v>$ is a back edge if and only if $d[v] \le d[u] < f[u] \le f[v]$.
- ***Forward edges*** are those edges $<u,v>$ connecting a vertex $u$ to a descendant $v$ in $G_T$. A edge $<u,v>$ is a forward edge if and only if $d[u] < d[v] < f[v] < f[u]$.
- ***Cross edges*** are all other edges.
  A node $u$ is in $DFSP(v)$ if and only if $d[u] \le d[v] < f[v] \le f[u]$.

Given a CFG $G=(N,E,h)$, a ***strongly connected region (SCR)*** is a nonempty set of nodes $S \subseteq N$, for which, given any $q,r \in S$, there exists a path from $q$ to $r$ and from $r$ to $q$. A SCR is a ***maximal SCR*** if none of its proper supersets is a SCR.

### 3.2 Definitions About Loops

We present brief descriptions of definitions about loops. The detailed version can be found in [7].

***Loops*** include ***outermost loops*** and ***inner loops***.

An ***outermost loop*** is a maximal SCR with at least one internal edge.

In any particular depth-first search, the first node of a loop $L$ to be traversed is defined to be the ***header*** of the loop, i.e. for the header $h$, $d[h]$ is the minimum in all the nodes in $L$. The set of other nodes is defined to be the ***loop body***.

An ***inner loop*** nested inside a loop $L$ with header $h$ is an outermost loop with respect to the subgraph with node set $(L-\{h\})$.

***Loop-nesting forest*** is a data structure that represents the containment relation between loops in a control flow graph.[4]

Given a loop $L$ with header $h$ and an edge $<q,r>$, $q \notin L$, $r \in L-\{h\}$, then $r$ is called a ***re-entry*** of this loop, and $<q,r>$ is called a ***re-entry edge***.

For a node $n$ in a loop body, its ***innermost loop*** is the smallest loop containing $n$., and the header of this loop is called $n$'s ***innermost loop header***. The ***loop header list*** of $n$ consists of its innermost loop header $h_1$, $h_1$'s innermost loop header $h_2$, $h_2$'s innermost loop header $h_3$, and so on. The loop header list of $d$ in Fig.3 is $[c,b,a]$.

### 3.3 Properties of Loop and DFS

We discover that there are some interesting properties of DFS and loops defined in section 3.2, which are helpful in identifying loops.

***Ancestor Property:*** For any node $n$, all of its loop headers must be in $DFSP(n)$.

***Nesting Property:*** Two different loop headers $x,y$ of node $n$ must be nested, i.e. either $x$ is a loop header of $y$, or $y$ is a loop header of $x$.

***Direct Transitive Property:*** Given that node $m$ is a child of node $n$ in the $DFST$, a loop header $x$ of $m$ is also a loop header of $n$ if and only if $x \neq m$.

***Indirect Transitive Property:*** Given that node $\mathbf{m}$ is a successor of node $\mathbf{n}$ and $\mathbf{m} \notin \mathbf{DFSP(n)}$, a loop header $\mathbf{x}$ of $\mathbf{m}$ is also a loop header of $\mathbf{n}$ if and only if $\mathbf{x} \in \mathbf{DFSP(n)}$.

All these properties can be proved, and here are some lemmas used during the proof:

***Lemma 1:*** Given an edge $\mathbf{<n,m>}$, if $\mathbf{d[n]<d[m]}$, then $\mathbf{f[n]>f[m]}$.

***Lemma 2:*** Given an edge $\mathbf{<n,m>}$, if it is not a back edge, then $\mathbf{f[n]>f[m]}$.

***Lemma 3:*** Given a re-entry $\mathbf{<n,m>}$ of loop $\mathbf{L}$, if the header of $\mathbf{L}$ is $\mathbf{x}$, then $\mathbf{f[n]>f[x]>f[m]}$.

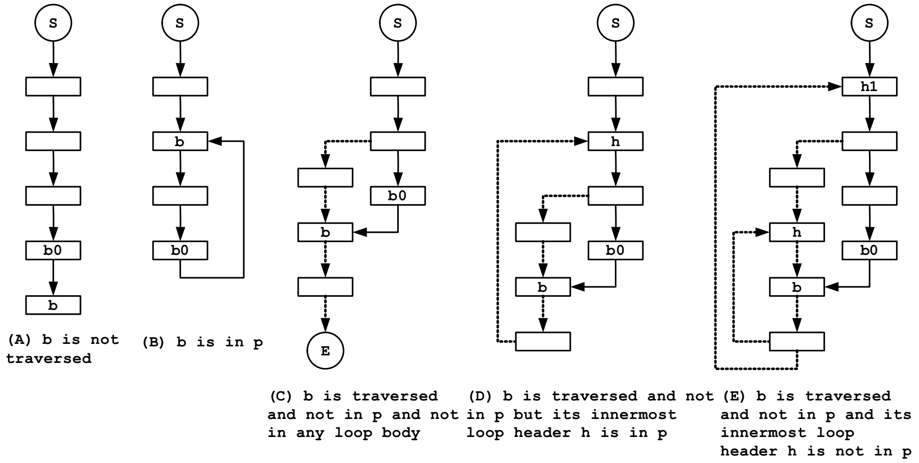
## 4 Algorithm for Identifying Loops

Statement of the problem: Given a CFG $\mathbf{G=(N,E,h_0)}$, for each node $\mathbf{n} \in \mathbf{N}$:

(1) Decide whether or not $\mathbf{n}$ is a loop header;
(2) Decide whether or not $\mathbf{n}$ is in a loop body; If yes, which node is its innermost loop header?
(3) Decide whether or not $\mathbf{n}$ is a re-entry; If yes, which edges are the re-entry edges?

Based on these properties given in section 3.3, we propose a new algorithm which contains two parts: traversing a CFG based on depth-first search, and tagging loop headers on demand.

In contrast with the multi-pass algorithms proposed by Tarjan, Havlak and Ramalingam, our algorithm collects and propagates loop header information during depth-first search based on these properties, so it doesn't need the second bottom-up traversal based on UNION-FIND operations to do the same thing.



(A) b is not traversed

(B) b is in p

(C) b is traversed and not in p and not in any loop body

(D) b is traversed and not in p but its innermost loop header h is in p

(E) b is traversed and not in p and its innermost loop header h is not in p

**Fig. 4.** Cases during traversing

**Traversing:** The algorithm visits all the nodes in $N$, starting from $h_0$ recursively in depth-first search order. When a node $b_0$ is visited, it is marked as ***traversed*** and let $p$ be the current path from $h_0$ to $b_0$ in the depth-first spanning tree (i.e. $DFSP(b_0)$). Each successor $b$ of $b_0$ is checked in turn as followed:

(A) If $b$ is a new node, i.e. $b$ is not ***traversed*** yet, as shown in Fig.4(A): traverse it recursively; if it is found in a loop body after being traversed, tag $b$'s innermost loop header as a loop header of $b_0$ if the header node is in $p$;

(B) If $b$ is ***traversed*** already, and it is in $p$, as shown in Fig.4(B): mark $b$ as a loop header, and tag $b$ as a loop header of $b_0$;

(C) If $b$ is ***traversed*** already, and it is not in $p$, or any loop body, as shown in Fig.4(C): just skip it;

(D) If $b$ is ***traversed*** already, and it is not in $p$, but it is in a loop body whose innermost loop header $h$ is in $p$, as shown in Fig.4(D): tag $h$ as a loop header of $b_0$;

(E) If $b$ is ***traversed*** already, and it is not in $p$, but it is in a loop body whose innermost loop header is not in $p$, as shown in Fig.4(E): mark $b$ as a re-entry node, and mark $<b_0,b>$ as a re-entry edge. Find the innermost loop header $h_1$ of $b$ in $p$ if it exists, then tag $h_1$ as a loop header of $b_0$.

The pseudo code of traversing is shown as following:

```
procedure identify_loops(CFG G=(N,E,h0)):
   foreach(Block b in N): // init
      initialize(b); // zeroize flags & properties
   trav_loops_DFS(h0,1);

function trav_loops_DFS(Block b0, int DFSP_pos):
//return: innermost loop header of b0
   Mark b0 as traversed;
   b0.DFSP_pos := DFSP_pos;//Mark b0's position in DFSP
   foreach(Block b in Succ(b0)):
      if(b is not traversed):
         // case(A), new
         Block nh := trav_loops_DFS(b, DFSP_pos+1);
         tag_lhead(b0, nh);
      else:
         if(b.DFSP_pos > 0): // b in DFSP(b0)
            // case(B)
            Mark b as a loop header;
            tag_lhead(b0, b);
         else if(b.iloop_header == nil):
            // case(C), do nothing
         else:
            Block h := b.iloop_header;
            if(h.DFSP_pos > 0): // h in DFSP(b0)
               // case(D)
               tag_lhead(b0, h);
            else: // h not in DFSP(b0)
```

```
               // case(E), reentry
               Mark b and (b0,b) as re-entry;
               Mark the loop of h as irreducible;
               while(h.iloop_header!=nil):
                  h := h.iloop_header;
                  if(h.DFSP_pos > 0): // h in DFSP(b0)
                     tag_lhead(b0, h);
                     break;
                  Mark the loop of h as irreducible;
   b0.DFSP_pos := 0; // clear b0's DFSP position
   return b0.iloop_header;
```

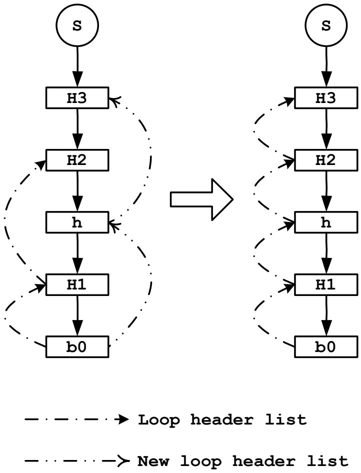


**Fig. 5.** Tagging loop headers

**Tagging:** When tagging ***h*** as a loop header of $\boldsymbol{b_0}$, weave ***h*** and its loop header list (if exists) into the current loop header list of $\boldsymbol{b_0}$ according to their positions in ***p***, as shown in Fig.5. The pseudo code of tagging loop headers is shown as following.

```
procedure tag_lhead(Block b, Block h):
   if(b == h or h == nil) return;
   Block cur1 := b, cur2 := h;
   while(cur1.iloop_header!=nil):
      Block ih := cur1.iloop_header;
      if(ih == cur2) return;
      if(ih.DFSP_pos < cur2.DFSP_pos):
         cur1.iloop_header := cur2;
         cur1 := cur2;
         cur2 := ih;
      else:
         cur1 := ih;
   cur1.iloop_header := cur2;
```

Based on properties given in section 3.3, it can be proved that the loop header of every node will be correctly tagged after this one-pass DFS traversing.

## 5 Complexity Analysis and Unstructuredness Coefficient

We now discuss the complexity of the algorithm given in section 4.

Given a CFG $G=(N,E,h_0)$, **trav_loops_DFS** is called recursively for each node $n \in N$ exactly one time, so it is called $N$ times totally.

In one invocation of **trav_loops_DFS**, the **foreach** loop is executed for each out-edge of the current node. It follows that in all invocations of **trav_loops_DFS** for all nodes in $G$, the **foreach** loop is executed exactly one time for each edge in $E$, so it is executed $E$ times totally.

In the $i$*-th* execution of the **foreach** loop ($i \in [1,E]$), only one of case (A), (B), (C), (D) and (E) can be chosen. Let $x_i$ be the execution times of the **while** loop in **trav_loops_DFS**, and let $y_i$ be the execution times of the **while** loop in **tag_lhead**. The complexity of case (A) is $O(1+y_i)$, except for the recursive call to **trav_loops_DFS** which is counted in the above already; the complexity of case (B) and (D) is $O(1+y_i)$; the complexity of case (C) is $O(1)$; the complexity of case (E) is $O(1+x_i+y_i)$. Notice that in case (A), (B), (C) and (D), $x_i=0$, and in case (C) $y_i=0$ too.

In summary, the total complexity of the algorithm is $O(N+E+\Sigma x_i+\Sigma y_i)$, $i \in [1,E]$. Let $k=1+(\Sigma x_i+\Sigma y_i)/E$, it follows that the total complexity can be expressed as $O(N+k*E)$.

In the following, we discuss the meaning of $x_i$, $y_i$ and $k$ behind these mathematical expressions.

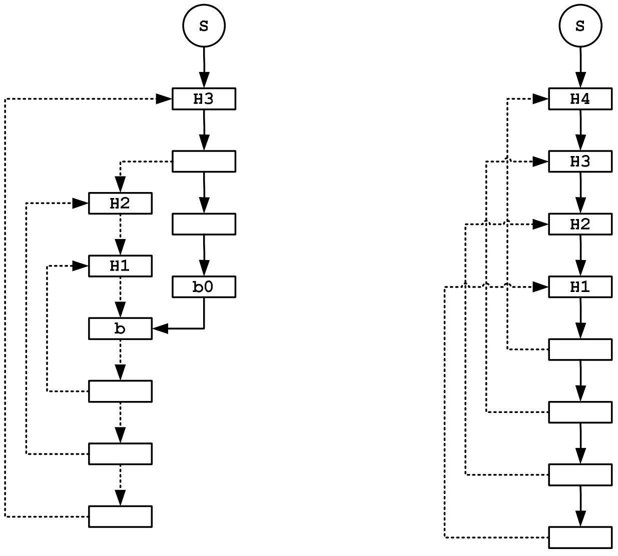
As shown in Fig.6(A), the **while** loop in **trav_loops_DFS** is executed because the edge $<b_0,b>$ skips multi level loop headers and jumps directly into the $(x_i+1)$*th* inner loop. This situation is called multientry unstructuredness[14].

As shown in Fig.6(B), the **while** loop in **tag_lhead** is executed mainly because the back edges of loops overlap with each other, and $y_i$ is the rough measurement of overlapping levels. This situation is called overlapping unstructuredness[14].

Multientry unstructuredness is irreducible, whereas overlapping unstructuredness is reducible. Multientry unstructuredness is caused by forward edges while overlapping unstructuredness is caused by backward edges. They both contribute to the total unstructuredness of a CFG.

$k=1+(\Sigma x_i+\Sigma y_i)/E$, it describes the ratio of the total unstructuredness, including both multientry unstructuredness and overlapping unstructuredness, to the size of a CFG. Hence we call $k$ the ***unstructuredness coefficient***.

Please notice that the ***unstructuredness coefficient*** $k$ is usually small: In today's binary executables, unstructuredness is introduced mostly by optimization compilers, not by programmers instead. The main reason is that structured programming has been well adopted already. In addition, unstructured code is hard to maintain correctness, even introduced by compilers. Therefore, although unstructuredness can be found in almost every binary code, the majority of binary code is well structured. Experiments in the next section validate such analysis.



(A) Multientry unstructuredness

(E) Overlapping unstructuredness

**Fig. 6.** Unstructuredness of loops

## 6 Experimental Results

We analyze binary executables on different operating systems using ***BESTAR*** (Binary Executable Structurizer and Analyzer), our in-house decompiler which has implemented the algorithms described above, including our algorithm, the Havlak-Tarjan algorithm and the Ramalingam-Havlak-Tarjan algorithm.

The selected instances include: 1). System binary executables of Windows XP, including **kernel32.dll**, **user32.dll** and **explorer.exe**; 2) Well-known applications on Linux, including **samba 3.0.23d**, **sendmail 8.13.8** and **vsftpd 2.0.5**, which are compiled by "**gcc -O2**".

Table.1 shows the statistics about loops in these instances identified by algorithms. There are totally 11526 CFGs in these instances. In these CFGs, there are 174 irreducible CFGs and 7841 loops. The experimental results of all these algorithms are the same, which validate the correctness of our algorithm and its implementation.

A phenomenon we have discovered from these results is that all these instances contain irreducible CFGs.

Another important phenomenon is that $\boldsymbol{k}$ is small in all these instances. Table.2 and Fig.7 show statistics of $\boldsymbol{k}$ with respect to the number of nodes of CFGs in these instances. The statistics show that in these real-world instances the unstructuredness coefficient is usually smaller than 1.5 and its average value is hardly correlated to the size of CFG. We call this phenomenon **"low-unstructuredness"** property of CFGs, and this property distinguishes real-world CFGs from general single-root connected directed graphs.

**Table 1.** Statistics of loops

| | e ne | use | exp e | s | sen<br>1 | vs tp |
|---|---|---|---|---|---|---|
| Gs | 1 | 1711 | 1 | 9 | | 9 |
| e uc e Gs | | | | 1 | 71 | 7 |
| p e e s | 11 | | | 11 | 1 | 1 |
| v | 1 1 | 1 1 | 1 1 | 1 1 | 1 | 1 1 |
| x | 1 | 1 9 | 1 1 | 1 | 1 1 | 1 1 |
| n | 1 | 1 | 1 | 1 | 1 | 1 |

**Table 2.** Statistics of k to N

| | Ma (k) | (k) | Mi (k) |
|---|---|---|---|
| 49 | | | |
| 99 | 4 | | |
| 49 | 4 | | |
| 99 | | 6 | |
| 49 | | 6 | |
| 99 | | | |
| 49 | | | |
| 99 | 9 | | |

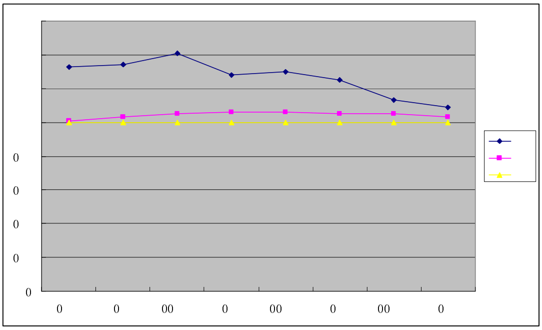


**Fig. 7.** Statistics of k to N

For performance comparison, we ran all these algorithms on an unloaded 2.6GHz AMD Opteron Server, with each implementation properly optimized. Table.3 and Fig.8 show the time spent during processing these instances. The results show that our algorithm is 2-5 times faster than the Havlak-Tarjan algorithm, and 2-8 times faster than the Ramalingam-Havlak-Tarjan algorithm.

**Table 3.** Time of algorithms(in μsec, lower is better)

| | ke e | use | e o e | sa a | se ai | sf |
|---|---|---|---|---|---|---|
| u | | 6 6 | 4 | 6 | | |
| Ha ak | 44 | | | 6 | | |
| a a i a | | 4 | | | 9 6 | 4 |

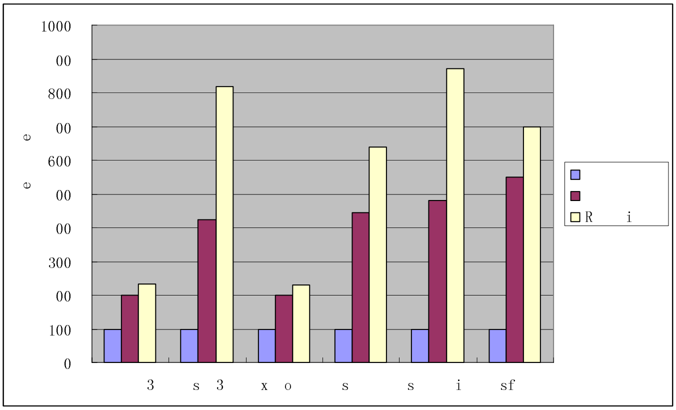


**Fig. 8.** Time comparison of algorithms (lower is better)

Here is another interesting result: although the Ramalingam-Havlak-Tarjan algorithm is an "improved" version of the Havlak-Tarjan algorithm, its performance is even worse than the latter. Based on the "low-unstructuredness" property of CFGs, this phenomenon can be easily explained: in order to reach almost linear time complexity when the target CFG's multientry unstructuredness is very high, Ramalingam adds extra procedures to solve least common ancestors and to mark irreducible loops, and both need UNION-FIND operations; however, for real-world CFGs, the unstructuredness is low, so the extra procedures contribute little to the performance and slow down the whole process instead.

## 7 Conclusion

This paper presents an innovative method for identifying loops in binary executables. First, we explore some useful properties of loops and DFS which make DFS collecting more information than simple forward/cross/backward edge information. Then, we propose the algorithm building on a one-pass DFS traversal and these properties. It does not use any complicated data structures such as Interval/DSG or UNION-FIND sets, so it is simpler and easier to implement than classical multi-pass traversal algorithms.

The complexity of our method is $O(N+k*E)$, where $k$ is the ***unstructuredness coefficient***, a new concept proposed in this paper to describe the unstructuredness of CFGs.

The ***unstructuredness coefficient*** $k$ is usually small, because structured programming has been well adopted, and unstructured code is hard to maintain its correctness, even introduced by compilers. Hence although unstructuredness can be found in almost every binary code, the majority of binary code is well structured. In fact, we found that in real-world binaries the average value of $k$ is usually smaller than 1.5 and hardly correlated to the size of CFGs. Such "low-unstructuredness" property distinguishes these CFGs from general single-root connected directed graphs, and it offers an explanation of why those algorithms with low time complexity on arbitrary graphs perform not quite well on "real" CFGs.

Using ***BESTAR*** (Binary Executable Structurizer and Analyzer), our in-house decompiler, we have applied the algorithm and classical algorithms to 11526 CFGs in 6 typical binary executables on Windows XP and Linux. Due to the simplicity of our algorithm and the "low-unstructuredness" property of real-world binaries, our algorithm is 2-5 times faster than the Havlak-Tarjan algorithm[7], and 2-8 times faster than the Ramalingam-Havlak-Tarjan algorithm[3].

Due to its remarkable performance, our algorithm could also be used in other applications, besides general decompilation, such as computing the SSA form or sequentializing program dependence graphs during just-in-time compilation.

## References

[1] http://www.program-transformation.org/Transform/HistoryOfDecompilation1

[2] Muchnick, S.S.: Advanced Compiler Design and Implementation. Elsevier Science, Amsterdam (1997)

[3] Ramalingam, G.: Identifying loops in almost linear time. ACM Transactions on Programming Languages and Systems 21(2) (1999)

[4] Ramalingam, G.: On loops, dominators, and dominance frontiers, ACM Transactions on Programming Languages and Systems 24(5) (2002)

[5] Allen, F.E.: Control flow analysis. SIGPLAN Notices 5(7), 1–19 (1970)

[6] Cocke, J.: Global common subexpression elimination. SIGPLAN Notices 5(7), 20–25 (1970)

[7] Havlak, P.: Nesting of reducible and irreducible loops. ACM Transactions on Programming Languages and Systems 19(4) (1997)

[8] Tarjan, R.E.: Testing flow graph reducibility, J. Comput. Syst. Sci. 9 (1974)

[9] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge (2001)

[10] Cooper, K.D., Harvey, T.J., Kennedy, K.: A Simple Fast Dominance Algorithm, Software Practice and Experience (2001)

[11] Hecht, M.S., Ullman, J.D.: Flow graph reducibility. SIAM Journal of Computing 1(2), 188–202 (1972)

[12] Steensgaard, B.: Sequentializing program dependence graphs for irreducible programs. Tech. Rep. MSR-TR-93-14, Microsoft Research, Redmond, Wash (1993)

[13] Sreedhar, V.C., Gao, G.R., Lee, Y.F.: Identifying loops using DJ graphs. ACM Transactions on Programming Languages and Systems 18(6) (1996)

[14] Cifuentes, C.: Reverse compilation techniques. PhD Thesis, Queensland University of Technology (1994)

# Accelerated Data-Flow Analysis

Jérôme Leroux and Grégoire Sutre

LaBRI, Université de Bordeaux, CNRS
Domaine Universitaire, 351, cours de la Libération, 33405 Talence, France
{leroux,sutre}@labri.fr

**Abstract.** Acceleration in symbolic verification consists in computing the exact effect of some control-flow loops in order to speed up the iterative fix-point computation of reachable states. Even if no termination guarantee is provided in theory, successful results were obtained in practice by different tools implementing this framework. In this paper, the acceleration framework is extended to data-flow analysis. Compared to a classical widening/narrowing-based abstract interpretation, the loss of precision is controlled here by the choice of the abstract domain and does not depend on the way the abstract value is computed. Our approach is geared towards precision, but we don't loose efficiency on the way. Indeed, we provide a cubic-time acceleration-based algorithm for solving interval constraints with full multiplication.

## 1 Introduction

Model-checking safety properties on a given system usually reduces to the computation of a precise enough invariant of the system. In traditional symbolic verification, the set of all reachable (concrete) configurations is computed iteratively from the initial states by a standard fix-point computation. This reachability set is the most precise invariant, but quite often (in particular for software systems) a much coarser invariant is sufficient to prove correctness of the system. Data-flow analysis, and in particular abstract interpretation [CC77], provides a powerful framework to develop analysis for computing such approximate invariants.

A data-flow analysis of a program basically consists in the choice of a (potentially infinite) complete lattice of data properties for program variables together with transfer functions for program instructions. The merge over all path (MOP) solution, which provides the most precise abstract invariant, is in general over-approximated by the minimum fix-point (MFP) solution, which is computable by Kleene fix-point iteration. However the computation may diverge and *widening/narrowing operators* are often used in order to enforce convergence at the expense of precision [CC77, CC92]. While often providing very good results, the solution computed with widenings and narrowings may not be the MFP solution. This may lead to abstract invariants that are too coarse to prove safety properties on the system under check.

Techniques to help convergence of Kleene fix-point iterations have also been investigated in symbolic verification of infinite-state systems. In these works, the

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 184–199, 2007.

objective is to compute the (potentially infinite) reachability set for automata with variables ranging over unbounded data, such as counters, clocks, stacks or queues. So-called *acceleration* techniques (or *meta-transitions*) have been developped [BW94, BGWW97, CJ98, FIS03, FL02] to speed up the iterative computation of the reachability set. Basically, acceleration consists in computing in one step the effect of iterating a given loop (of the control flow graph). Accelerated symbolic model checkers such as Lash [Las], TReX [ABS01], and Fast [BFLP03] successfully implement this approach.

*Our contribution.* In this paper, we extend acceleration techniques to data-flow analysis and we apply these ideas to interval analysis. Acceleration techniques for (concrete) reachability set computations may be equivalently formalized "semantically" in terms of control-flow path languages [LS05] or "syntactically" in terms of control-flow graph unfoldings [BFLS05]. We extend these concepts to the MFP solution in a generic data-flow analysis framework, and we establish several links between the resulting notions. It turns out that, for data-flow analysis, the resulting "syntactic" notion, based on graph *flattenings*, is more general that the resulting "semantic" notion, based on restricted regular expressions. We then propose a generic flattening-based semi-algorithm for computing the MFP solution. This semi-algorithm may be viewed as a generic template for applying acceleration-based techniques to constraint solving.

We then show how to instantiate the generic flattening-based semi-algorithm in order to obtain an efficient constraint solver[1] for integers, for a rather large class of constraints using addition, (monotonic) multiplication, factorial, or any other *bounded-increasing* function. The intuition behind our algorithm is the following: we propagate constraints in a breadth-first manner as long as the least solution is not obtained, and variables involved in a "useful" propagation are stored in a graph-like structure. As soon as a cycle appears in this graph, we compute the least solution of the set of constraints corresponding to this cycle. It turns out that this acceleration-based algorithm always terminates in cubic-time.

As the main result of the paper, we then show how to compute in cubic-time the least solution for interval constraints with full addition and multiplication, and intersection with a constant. The proof uses a least-solution preserving translation from interval constraints to the class of integer constraints introduced previously.

*Related work.* In [Kar76], Karr presented a polynomial-time algorithm that computes the set of all affine relations that hold in a given control location of a (numerical) program. Recently, the complexity of this algorithm was revisited in [MOS04] and a fine upper-bound was presented. For interval constraints with affine transfer functions, the exact least solution may be computed in cubic-time [SW04]. Strategy iteration was proposed in [CGG$^+$05] to speed up Kleene fix-point iteration with better precision than widenings and narrowings, and this

[1] By solver, we mean an algorithm computing the least solution of constraint systems.

approach has been developped in [TG07] for interval constraint solving with full addition, multiplication and intersection. Strategy iteration may be viewed as an instance of our generic flattening-based semi-algorithm. The class of interval constraints that we consider in this paper contains the one in [SW04] (which does not include interval multiplication) but it is more restrictive than the one in [TG07]. We are able to maintain the same cubic-time complexity as in [SW04], and it is still an open problem whether interval constraint solving can be performed in polynomial-time for the larger class considered in [TG07].

*Outline.* The paper is organized as follows. Section 2 presents our acceleration-based approach to data-flow analysis. We then focus on interval constraint-based data-flow analysis. We present in section 3 a cubic-time algorithm for solving a large class of constraints over the integers, and we show in section 4 how to translate interval constraints (with multiplication) into the previous class of integer constraints, hence providing a cubic-time algorithm for interval constraints. Section 5 presents some ideas for future work. Please note that due to space constraints, most proofs are only sketched in this paper. A long version of the paper with detailed proofs can be obtained from the authors.

## 2 Acceleration in Data Flow Analysis

This section is devoted to the notion of acceleration in the context of data-flow analysis. Acceleration techniques for (concrete) reachability set computations [BW94, BGWW97, CJ98, FIS03, FL02, LS05, BFLS05] may be equivalently formulated in terms of control-flow path languages or control-flow graph unfoldings. We shall observe that this equivalence does not hold anymore when these notions are lifted to data-flow analysis. All results in this section can easily be derived from the definitions, and they are thus presented without proofs.

### 2.1 Lattices, Words and Graphs

We respectively denote by and the usual sets of nonnegative integers and integers. For any set $S$, we write $(S)$ for the set of subsets of $S$. The *identity* function over $S$ is written ${}_S$, and shortly when the set $S$ is clear from the context.

Recall that a *complete lattice* is any partially ordered set $(A, \sqsubseteq)$ such that every subset $X \subseteq A$ has a *least upper bound* $\bigsqcup X$ and a *greatest lower bound* $\bigsqcap X$. The *supremum* $\bigsqcup A$ and the *infimum* $\bigsqcap A$ are respectively denoted by $\top$ and $\bot$. A function $f \in A \to A$ is *monotonic* if $f(x) \sqsubseteq f(y)$ for all $x \sqsubseteq y$ in $A$. Recall that from Knaster-Tarski's Fix-point Theorem, any monotonic function $f \in A \to A$ has a *least fix-point* given by $\bigsqcap \{a \in A \mid f(a) \sqsubseteq a\}$. For any monotonic function $f \in A \to A$, we denote by $f^*$ the monotonic function in $A \to A$ defined by $f^*(x) = \bigsqcap \{a \in A \mid (x \sqcup f(a)) \sqsubseteq a\}$, in other words $f^*(x)$ is the least fix-point of $f$ greater than $x$.

For any complete lattice $(A, \sqsubseteq)$ and any set $S$, we also denote by $\sqsubseteq$ the partial order on $S \to A$ defined as the point-wise extension of $\sqsubseteq$, i.e. $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all $x \in S$. The partially ordered set $(S \to A, \sqsubseteq)$ is also a complete lattice, with lub $\bigsqcup$ and glb $\bigsqcap$ satisfying $(\bigsqcup F)(s) = \bigsqcup\{f(s) \mid f \in F\}$ and $(\bigsqcap F)(s) = \bigsqcap\{f(s) \mid f \in F\}$ for any subset $F \subseteq S \to A$. Given any integer $n \geq 0$, we denote by $A^n$ the set of $n$-tuples over $A$. We identify $A^n$ with the set $\{1, \ldots, n\} \to A$, and therefore $A^n$ equipped with the point-wise extension of $\sqsubseteq$ also forms a complete lattice.

Let $\Sigma$ be an *alphabet* (a finite set of *letters*). We write $\Sigma^*$ for the set of all (finite) *words* $l_0 \cdots l_n$ over $\Sigma$, and $\varepsilon$ denotes the empty word. Given any two words $x$ and $y$, we denote by $x \cdot y$ (shortly written $xy$) their *concatenation*. A subset of $\Sigma^*$ is called a *language*.

A (directed) *graph* is any pair $G = (V, \to)$ where $V$ is a set of *vertices* and $\to$ is a binary relation over $V$. A pair $(v, v')$ in $\to$ is called an *edge*. A (finite) *path* in $G$ is any (non-empty) sequence $v_0, \ldots, v_k$ of vertices, also written $v_0 \to v_1 \cdots v_{k-1} \to v_k$, such that $v_{i-1} \to v_i$ for all $1 \leq i \leq k$. The nonnegative integer $k$ is called the *length* of the path, and the vertices $v_0$ and $v_k$ are respectively called the *source* and *target* of the path. A *cycle* on a vertex $v$ is any path of non-zero length with source and target equal to $v$. A cycle with no repeated vertices other than the source and the target is called *elementary*. We write $\xrightarrow{*}$ for the reflexive-transitive closure of $\to$. A *strongly connected component* (shortly *SCC*) in $G$ is any equivalence class for the equivalence relation $\overset{*}{\leftrightarrow}$ on $V$ defined by: $v \overset{*}{\leftrightarrow} v'$ if $v \xrightarrow{*} v'$ and $v' \xrightarrow{*} v$. We say that an SCC is *cyclic* when it contains a unique elementary cycle up to cyclic permutation.

## 2.2 Programs and Data-Flow Solutions

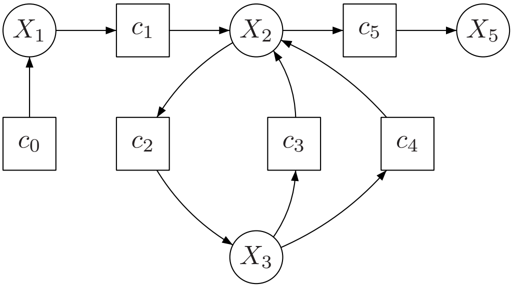
For the rest of this section, we consider a complete lattice $(A, \sqsubseteq)$. In our setting, a program will represent an instance (for some concrete program) of a data-flow analysis framework over $(A, \sqsubseteq)$. To simplify the presentation, we will consider programs given as unstructured collections of commands (this is not restrictive as control-flow may be expressed through variables).

Formally, assume a finite set $\mathcal{X}$ of *variables*. A *command* on $\mathcal{X}$ is any tuple $\langle X_1, \ldots, X_n; f; X\rangle$, also written $X := f(X_1, \ldots, X_n)$, where $n \in$  is an *arity*, $X_1, \ldots, X_n \in \mathcal{X}$ are pairwise disjoint *input variables*, $f \in A^n \to A$ is a monotonic *transfer function*, and $X \in \mathcal{X}$ is an *output variable*. Intuitively, a command $X := f(X_1, \ldots, X_n)$ assigns variable $X$ to $f(X_1, \ldots, X_n)$ and lets all other variables untouched. A *valuation* on $\mathcal{X}$ is any function $\rho$ in $\mathcal{X} \to A$. The *data-flow semantics* $[\![c]\!]$ of any command $c = \langle X_1, \ldots, X_n; f; X\rangle$ on $\mathcal{X}$ is the monotonic function in $(\mathcal{X} \to A) \to (\mathcal{X} \to A)$ defined by $[\![c]\!](\rho)(X) = f(\rho(X_1), \ldots, \rho(X_n))$ and $[\![c]\!](\rho)(Y) = \rho(Y)$ for all $Y \neq X$.

A *program* over $(A, \sqsubseteq)$ is any pair $\mathcal{P} = (\mathcal{X}, C)$ where $\mathcal{X}$ is a finite set of *variables* and $C$ is a finite set of commands on $\mathcal{X}$.

*Example 2.1.* Consider the C-style source code given on the left-hand side below, that we want to analyse with the complete lattice $(\mathcal{I}, \sqsubseteq)$ of intervals of  . The corresponding program $\mathcal{E}$ is depicted graphically on the right-hand side below.

```
1  x = 1;
2  while (x ≤ 100) {
3    if (x ≥ 50) x = x−3;
4    else x = x+2;
5  }
```



Formally, the set of variables of $\mathcal{E}$ is $\{X_1, X_2, X_3, X_5\}$, representing the value of the variable $x$ at program points 1, 2, 3 and 5. The set of commands of $\mathcal{E}$ is $\{c_0, c_1, c_2, c_3, c_4, c_5\}$, with:

$$\begin{array}{llll} c_0 : & X_1 := \top & c_3 : & X_2 := (X_3 \sqcap [50, +\infty]) - \{3\} \\ c_1 : & X_2 := (\{0\} \,.\, X_1) + \{1\} & c_4 : & X_2 := (X_3 \sqcap ] - \infty, 49]) + \{2\} \\ c_2 : & X_3 := X_2 \sqcap ] - \infty, 100] & c_5 : & X_5 := X_2 \sqcap [101, +\infty[ \end{array}$$

We will use language-theoretic terminology and notations for traces in a program. A *trace* in $\mathcal{P}$ is any word $c_1 \cdots c_k$ over $C$. The empty word $\varepsilon$ denotes the empty trace and $C^*$ denotes the set of all traces in $\mathcal{P}$. The data-flow semantics is extended to traces in the obvious way: $[\![\varepsilon]\!] =$ and $[\![c \cdot \sigma]\!] = [\![\sigma]\!] \circ [\![c]\!]$. Observe that $[\![\sigma \cdot \sigma']\!] = [\![\sigma']\!] \circ [\![\sigma]\!]$ for every $\sigma, \sigma' \in C^*$. We also extend the data-flow semantics to sets of traces by $[\![L]\!] = \bigsqcup_{\sigma \in L} [\![\sigma]\!]$ for every $L \subseteq C^*$. Observe that $[\![L]\!]$ is a monotonic function in $(\mathcal{X} \to A) \to (\mathcal{X} \to A)$, and moreover $[\![L_1 \cup L_2]\!] = [\![L_1]\!] \sqcup [\![L_2]\!]$ for every $L_1, L_2 \subseteq C^*$.

Given a program $\mathcal{P} = (\mathcal{X}, C)$ over $(A, \sqsubseteq)$, the *minimum fix-point solution* (*MFP-solution)* of $\mathcal{P}$, written $\Lambda_{\mathcal{P}}$, is the valuation defined as follows:

$$\Lambda_{\mathcal{P}} \;=\; \bigsqcap \; \{\rho \in \mathcal{X} \to A \mid [\![c]\!](\rho) \sqsubseteq \rho \text{ for all } c \in C\}$$

*Example 2.2.* The MFP-solution of the program $\mathcal{E}$ from Example 2.1 is the valuation:

$$\Lambda_{\mathcal{E}} \;=\; \{X_1 \mapsto \top,\ X_2 \mapsto [1, 51],\ X_3 \mapsto [1, 51],\ X_5 \mapsto \bot\}$$

Recall that we denote by $[\![C]\!]^*(\rho)$ the least fix-point of $[\![C]\!]$ greater than $\rho$. Therefore it follows from the definitions that $\Lambda_{\mathcal{P}} = [\![C]\!]^*(\bot)$. In our framework, the *merge over all paths solution* (*MOP-solution)* may be defined as the valuation $[\![C^*]\!](\bot)$, and the following proposition recalls well-known links between the MOP-solution, the MFP-solution and the ascending Kleene chain.

**Proposition 2.3.** *For any program $\mathcal{P} = (\mathcal{X}, C)$ over a complete lattice $(A, \sqsubseteq)$, we have:*

$$[\![C^*]\!](\bot) \quad \sqsubseteq \quad \bigsqcup_{k \in} [\![C]\!]^k(\bot) \quad \sqsubseteq \quad [\![C]\!]^*(\bot) \quad = \quad \Lambda_{\mathcal{P}}$$

### 2.3 Accelerability and Flattening

We now extend notions from accelerated symbolic verification to this data-flow analysis framework. Acceleration in symbolic verification was first introduced semantically, in the form of *meta-transitions* [BW94, BGWW97], which basically simulate the effect of taking a given control-flow loop arbitrarily many times. This leads us to the following proposition and definition.

**Proposition 2.4.** *Let $\mathcal{P} = (\mathcal{X}, C)$ denote a program over $(A, \sqsubseteq)$. For any languages $L_1, \ldots, L_k \subseteq C^*$, we have $(\llbracket L_k \rrbracket^* \circ \cdots \circ \llbracket L_1 \rrbracket^*)(\bot) \;\sqsubseteq\; \Lambda_{\mathcal{P}}$.*

**Definition 2.5.** *A program $\mathcal{P} = (\mathcal{X}, C)$ over a complete lattice $(A, \sqsubseteq)$ is called* MFP-accelerable *if $\Lambda_{\mathcal{P}} = (\llbracket \sigma_k \rrbracket^* \circ \cdots \circ \llbracket \sigma_1 \rrbracket^*)(\bot)$ for some words $\sigma_1, \ldots, \sigma_k \in C^*$.*

The following proposition shows that any program $\mathcal{P}$ for which the ascending Kleene chain stabilizes after finitely many steps is MFP-accelerable.

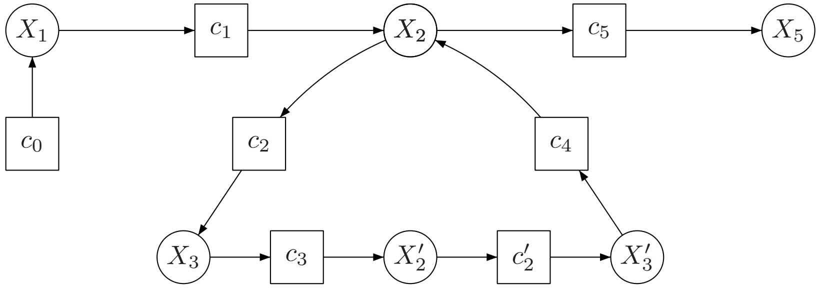
**Proposition 2.6.** *Let $\mathcal{P} = (\mathcal{X}, C)$ denote a program over $(A, \sqsubseteq)$. If we have $\llbracket C \rrbracket^k(\bot) = \Lambda_{\mathcal{P}}$ for some $k \in$ , then $\mathcal{P}$ is MFP-accelerable.*

Acceleration in symbolic verification was later expressed syntactically, in terms of flat graph unfoldings. When lifted to data-flow analysis, this leads to a more general concept than accelerability, and we will show that these two notions coincide for "concrete" programs (as in symbolic verification). We say that a program $\mathcal{P}$ is *single-input* if the arity of every command in $\mathcal{P}$ is at most 1.

Given a program $\mathcal{P} = (\mathcal{X}, C)$ over $(A, \sqsubseteq)$, an *unfolding* of $\mathcal{P}$ is any pair $(\mathcal{P}', \kappa)$ where $\mathcal{P}' = (\mathcal{X}', C')$ is a program and $\kappa \in \mathcal{X}' \to \mathcal{X}$ is a variable *renaming*, and such that $\langle \kappa(X'_1), \ldots, \kappa(X'_n); f; \kappa(X') \rangle$ is a command in $C$ for every command $\langle X'_1, \ldots, X'_n; f; X' \rangle$ in $C'$. The renaming $\kappa$ induces a Galois surjection $(\mathcal{X}' \to A, \sqsubseteq) \mathrel{\underset{\overrightarrow{\kappa}}{\overset{\overleftarrow{\kappa}}{\leftrightarrows}}} (\mathcal{X} \to A, \sqsubseteq)$ where $\overleftarrow{\kappa}$ and $\overrightarrow{\kappa}$ are defined as expected by $\overleftarrow{\kappa}(\rho) = \rho \circ \kappa$ and $\overrightarrow{\kappa}(\rho')(X) = \bigsqcup_{\kappa(X') = X} \rho'(X')$.

We associate a bipartite graph to any program in a natural way: vertices are either variables or commands, and edges denote input and output variables of commands. Formally, given a program $\mathcal{P} = (\mathcal{X}, C)$, the *program graph* of $\mathcal{P}$ is the labeled graph $G_{\mathcal{P}}$ where $\mathcal{X} \cup C$ is the set of vertices and with edges $(c, X)$ and $(X_i, c)$ for every command $c = \langle X_1, \ldots, X_n; f; X \rangle$ in $C$ and $1 \leq i \leq n$. We say that $\mathcal{P}$ is *flat* if there is no SCC in $G_{\mathcal{P}}$ containing two distinct commands with the same output variable. A *flattening* of $\mathcal{P}$ is any unfolding $(\mathcal{P}', \kappa)$ of $\mathcal{P}$ such that $\mathcal{P}'$ is flat.

*Example 2.7.* A flattening of the program $\mathcal{E}$ from Example 2.1 is given below. Intuitively, this flattening represents a possible unrolling of the while-loop where the two branches of the inner conditional alternate.



**Lemma 2.8.** *Let $\mathcal{P} = (\mathcal{X}, C)$ denote a program over $(A, \sqsubseteq)$. For any unfolding $(\mathcal{P}', \kappa)$ of $\mathcal{P}$, with $\mathcal{P}' = (\mathcal{X}', C')$, we have $\overrightarrow{\kappa} \circ [\![C']\!]^* \circ \overleftarrow{\kappa} \ \sqsubseteq\ [\![C]\!]^*$.*

**Proposition 2.9.** *Let $\mathcal{P} = (\mathcal{X}, C)$ denote a program over $(A, \sqsubseteq)$. For any unfolding $(\mathcal{P}', \kappa)$ of $\mathcal{P}$, we have $\overrightarrow{\kappa}(\Lambda_{\mathcal{P}'}) \ \sqsubseteq\ \Lambda_{\mathcal{P}}$.*

**Definition 2.10.** *A program $\mathcal{P} = (\mathcal{X}, C)$ over a complete lattice $(A, \sqsubseteq)$ is called* MFP-flattable *if $\Lambda_{\mathcal{P}} = \overrightarrow{\kappa}(\Lambda_{\mathcal{P}'})$ for some flattening $(\mathcal{P}', \kappa)$ of $\mathcal{P}$.*

Observe that any flat program is trivially MFP-flattable. The following proposition establishes links between accelerability and flattability. As a corollary to the proposition, we obtain that MFP-accelerability and MFP-flattability are equivalent for single-input programs.

**Proposition 2.11.** *The following relationships hold for programs over $(A, \sqsubseteq)$:*

*i) MFP-accelerability implies MFP-flattability.*
*ii) MFP-flattability implies MFP-accelerability for single-input programs.*

*Proof (Sketch).* To prove *i*), we use the fact that for every words $\sigma_1, \ldots, \sigma_k \in C^*$, there exists a finite-state automaton $\mathcal{A}$ without nested cycles recognizing $\sigma_1^* \cdots \sigma_k^*$. The "product" of any program $\mathcal{P}$ with $\mathcal{A}$ yields a flattening that "simulates" the effect of $\sigma_1^* \cdots \sigma_k^*$ on $\mathcal{P}$. To prove *ii*), we observe that for any flat single-input program $\mathcal{P}$, each non-trivial SCC of $G_{\mathcal{P}}$ is cyclic. We pick a "cyclic" trace (which is unique up to circular permutation) for each SCC, and we arrange these traces to prove that $\mathcal{P}$ is accelerable. Backward preservation of accelerability under unfolding concludes the proof. □

*Remark 2.12.* For any labeled transition system $\mathcal{S}$ with a set $S$ of states, the forward collecting semantics of $\mathcal{S}$ may naturally be given as a single-input program $\mathcal{P}_{\mathcal{S}}$ over $(\ (S), \subseteq)$. With respect to this translation (from $\mathcal{S}$ to $\mathcal{P}_{\mathcal{S}}$), the notion of flattability developped for accelerated symbolic verification of labeled transition systems coincide with the notions of MFP-accelerability and MFP-flattability defined above.

Recall that our main goal is to compute (exact) MFP-solutions using acceleration-based techniques. According to the previous propositions, flattening-based computation of the MFP-solution seems to be the most promising approach, and we will focus on this approach for the rest of the paper.

### 2.4 Generic Flattening-Based Constraint Solving

It is well known that the MFP-solution of a program may also be expressed as the least solution of a constraint system, and we will use this formulation for the rest of the paper. We will use some new terminology to reflect this new formulation, however notations and definitions will remain the same. A command $\langle X_1, \ldots, X_n; f; X\rangle$ will now be called a *constraint*, and will also be written $X \sqsupseteq f(X_1, \ldots, X_n)$. A program over $(A, \sqsubseteq)$ will now be called a *constraint system* over $(A, \sqsubseteq)$, and the MFP-solution will be called the *least solution*. Among all acceleration-based notions defined previously, we will only consider MFP-flattability for constraint systems, and hence we will shortly write *flattable* instead of MFP-flattable.

Given a constraint system $\mathcal{P} = (\mathcal{X}, C)$ over $(A, \sqsubseteq)$, any valuation $\rho \in \mathcal{X} \to A$ such that $\rho \sqsubseteq [\![C]\!](\rho)$ (resp. $\rho \sqsupseteq [\![C]\!](\rho)$) is called a *pre-solution* (resp. a *post-solution*). A post-solution is also shortly called a *solution*. Observe that the least solution $\varLambda_{\mathcal{P}}$ is the greatest lower bound of all solutions of $C$.

We now present a generic flattening-based semi-algorithm for constraint solving. Intuitively, this semi-algorithm performs a propagation of constraints starting from the valuation $\bot$, but at each step we extract a flat “subset” of constraints (possibly by duplicating some variables) and we update the current valuation with the least solution of this flat “subset” of constraints.

```
1  Solve($\mathcal{P} = (\mathcal{X}, C)$ : a constraint system)
2  $\rho \leftarrow \bot$
3  while $[\![C]\!](\rho) \not\sqsubseteq \rho$
4      construct a flattening $(\mathcal{P}', \kappa)$ of $\mathcal{P}$, where $\mathcal{P}' = (\mathcal{X}', C')$
5      $\rho' \leftarrow \rho \circ \kappa$
6      $\rho'' \leftarrow [\![C']\!]^*(\rho')$                    { $\overrightarrow{\kappa}(\rho'') \sqsubseteq [\![C]\!]^*(\rho)$ from Lemma 2.8 }
7      $\rho \leftarrow \rho \sqcup \overrightarrow{\kappa}(\rho'')$
8  return $\rho$
```

The Solve semi-algorithm may be viewed as a generic template for applying acceleration-based techniques to constraint solving. The two main challenges are (1) the construction of a suitable flattening at line 4, and (2) the computation of the least solution for flat constraint systems (line 6). However, assuming that all involved operations are effective, this semi-algorithm is *correct* (i.e. if it terminates then the returned valuation is the least solution of input constraint system), and it is *complete* for flattable constraint systems (i.e. the input constraint system is flattable if and only if there exists choices of flattenings at line 4 such that the while-loop terminates). We will show in the sequel how to instantiate the Solve semi-algorithm in order to obtain an efficient constraint solver for integers and intervals.

## 3 Integer Constraints

Following [SW04, TG07], we first investigate integer constraint solving in order to derive in the next section an interval solver. This approach is motivated by the encoding of an interval by two integers.

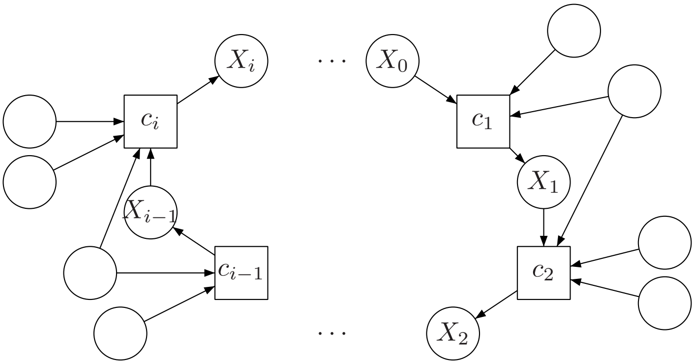
The *complete lattice of integers* $\mathcal{Z} = \ \cup \{-\infty, +\infty\}$ is equipped with the natural order:

$$-\infty < \cdots < -2 < -1 < 0 < 1 < 2 < \cdots < +\infty$$

Observe that the least upper bound $x \vee y$ and the greatest lower bound $x \wedge y$ respectively correspond to the maximum and the minimum. Addition and multiplication functions are extended from  to $\mathcal{Z}$ as in [TG07]:

$$\begin{array}{rclcl@{\quad}rclcl@{\quad}l}
x.0 & = & 0.x & = & 0 & \multicolumn{5}{c}{x + (-\infty) = (-\infty) + x = -\infty} & \text{for all } x \\
x.(+\infty) & = & (+\infty).x & = & +\infty & x.(-\infty) & = & (-\infty).x & = & -\infty & \text{for all } x > 0 \\
x.(+\infty) & = & (+\infty).x & = & -\infty & x.(-\infty) & = & (-\infty).x & = & +\infty & \text{for all } x < 0 \\
\multicolumn{5}{l}{x+(+\infty) = (+\infty) + x = +\infty} & & & & & & \text{for all } x > -\infty
\end{array}$$

A constraint system $\mathcal{P} = (\mathcal{X}, C)$ is said *cyclic* if the set of constraints $C$ is contained in a cyclic SCC. An example is given below.



Observe that a cyclic constraint system is flat. A *cyclic flattening* $(\mathcal{P}', \kappa)$ where $\mathcal{P}' = (\mathcal{X}', C')$ can be naturally *associated* to any cycle $X_0 \to c_1 \to X_1 \cdots \to c_n \to X_n = X_0$ of a constraint system $\mathcal{P}$, by considering the set $\mathcal{X}'$ of variables obtained from $\mathcal{X}$ by adding $n$ new copies $Z_1, \ldots, Z_n$ of $X_1, \ldots, X_n$ with the corresponding renaming $\kappa$ that extends the identity function over $\mathcal{X}$ by $\kappa(Z_i) = X_i$, and by considering the set of constraints $C' = \{c'_1, \ldots, c'_n\}$ where $c'_i$ is obtained from $c_i$ by renaming the output variable $X_i$ by $Z_i$ and by renaming the input variable $X_{i-1}$ by $Z_{i-1}$ where $Z_0 = Z_n$.

In section 3.1, we introduce an instance of the generic Solve semi-algorithm that solves constraint systems that satisfy a property called *bounded-increasing.* This class of constraint systems is extended in section 3.2 with test constraints allowing a natural translation of *interval* constraint systems to contraint systems in this class.

### 3.1 Bounded-Increasing Constraint Systems

A monotonic function $f \in \mathcal{Z}^k \rightarrow \mathcal{Z}$ is said *bounded-increasing* if for any $x_1 < x_2$ such that $f(\bot) < f(x_1)$ and $f(x_2) < f(\top)$ we have $f(x_1) < f(x_2)$. Intuitively $f$ is increasing over the domain of $x \in \mathcal{Z}^k$ such that $f(x) \notin \{f(\bot), f(\top)\}$.

*Example 3.1.* The guarded identity $x \mapsto x \wedge b$ where $b \in \mathcal{Z}$, the addition $(x, y) \mapsto x + y$, the two multiplication functions $\mathsf{mul}_+$ and $\mathsf{mul}_-$ defined below, the power by two $x \mapsto 2^{x \vee 0}$, the factorial $x \mapsto !(x \vee 1)$ are bounded-increasing. However the minimum and the maximum functions are not bounded-increasing.

$$\mathsf{mul}_+(x,y) = \begin{cases} x.y & \text{if } x, y \geq 0 \\ 0 & \text{otherwise} \end{cases} \qquad \mathsf{mul}_-(x,y) = \begin{cases} -x.y & \text{if } x, y < 0 \\ 0 & \text{otherwise} \end{cases}$$

A *bounded-increasing constraint* is a constraint of the form $X \geq f(X_1, \ldots, X_k)$ where $f$ is a bounded-increasing function. Such a constraint is said *upper-saturated* (resp. *lower-saturated*) by a valuation $\rho$ if $\rho(X) \geq f(\top)$ (resp. $\rho(X) \leq f(\bot)$). Given a constraint system $\mathcal{P} = (\mathcal{X}, C)$ and a bounded-increasing constraint $c \in C$ upper-saturated by a valuation $\rho_0$, observe that $[\![C]\!]^*(\rho_0) = [\![C']\!]^*(\rho_0)$ where $C' = C \backslash \{c\}$. Intuitively, an upper-saturated constraint for $\rho_0$ can be safely removed from a constraint system without modifying the least solution greater than $\rho_0$. The following lemma will be useful to obtain upper-saturated constraints.

**Lemma 3.2.** *Let $\mathcal{P}$ be a cyclic bounded-increasing constraint system. If $\rho_0$ is a pre-solution of $\mathcal{P}$ that does not lower-saturate any constraint, then either $\rho_0$ is a solution or $[\![C]\!]^*(\rho_0)$ upper-saturates a constraint.*

*Proof. (Sketch).* Let $X_0 \rightarrow c_1 \rightarrow X_1 \rightarrow \cdots \rightarrow c_n \rightarrow X_n = X_0$ be the unique (up to a cyclic permutation) cycle in the graph associated to $\mathcal{P}$. Consider a pre-solution $\rho_0$ of $\mathcal{P}$ that is not a solution. Let us denote by $(\rho_i)_{i \geq 0}$ the sequence of valuations defined inductivelly by $\rho_{i+1} = \rho_i \vee [\![C]\!](\rho_i)$. There are two cases:

- either there exists $i \geq 0$ such that $\rho_i$ upper-saturates a constraint $c_j$. Since $\rho_i \leq [\![C]\!]^*(\rho_0)$, we deduce that $[\![C]\!]^*(\rho_0)$ upper-saturates $c_j$.
- or $c_1, \ldots, c_n$ are not upper-saturated by any of the $\rho_i$. As these constraints are bounded-increasing, the sequence $(\rho_i)_{i \geq 0}$ is strictly increasing. Thus $(\bigvee_{i \geq 0} \rho_i)(X_j) = +\infty$ for any $1 \leq j \leq n$. Since $\bigvee_{i \geq 0} \rho_i \leq [\![C]\!]^*(\rho_0)$, we deduce that $[\![C]\!]^*(\rho_0)$ upper-saturates $c_1, \ldots, c_n$.

In both cases, $[\![C]\!]^*(\rho_0)$ upper-saturates at least one constraint. □

```
CyclicSolve ($\mathcal{P} = (\mathcal{X}, C)$ : a cyclic bounded−increasing constraint system,
             $\rho_0$ : a valuation)
 let $X_0 \rightarrow c_1 \rightarrow X_1 \cdots \rightarrow c_n \rightarrow X_n = X_0$ be the ‘‘unique’’ elementary cycle
 $\rho \leftarrow \rho_0$
 for $i = 1$ to $n$ do
       $\rho \leftarrow \rho \vee [\![c_i]\!](\rho)$
```

```
7   for $i = 1$ to $n$ do
8       $\rho \leftarrow \rho \vee [\![c_i]\!](\rho)$
9   if $\rho \geq [\![C]\!](\rho)$
10      return $\rho$
11  for $i = 1$ to $n$ do
12      $\rho(X_i) \leftarrow +\infty$
13  for $i = 1$ to $n$ do
14      $\rho \leftarrow \rho \wedge [\![c_i]\!](\rho)$
15  for $i = 1$ to $n$ do
16      $\rho \leftarrow \rho \wedge [\![c_i]\!](\rho)$
17  return $\rho$
```

**Proposition 3.3.** *The algorithm* CyclicSolve *returns* $[\![C]\!]^*(\rho_0)$ *for any cyclic constraint system* $\mathcal{P}$ *and for any valuation* $\rho_0$.

*Proof. (Sketch).* The first two loops (lines 5–8) propagate the valuation $\rho_0$ along the cycle two times. If the resulting valuation is not a solution at this point, then it is a pre-solution and no constraint is lower-saturated. From Lemma 3.2, we get that $[\![C]\!]^*(\rho_0)$ upper-saturates some constraint. Observe that the valuation $\rho$ after the third loop (lines 11–12) satisfies $[\![C]\!]^*(\rho_0) \sqsubseteq \rho$. The descending iteration of the last two loops yields (at line 17) $[\![C]\!]^*(\rho_0)$. □

We may now present our cubic time algorithm for solving bounded-increasing constraint systems. The main loop of this algorithm first performs $|C|+1$ rounds of Round Robin iterations and keeps track for each variable of the last constraint that updated its value. This information is stored in a partial function $\lambda$ from $\mathcal{X}$ to $C$. The second part of the main loop checks whether there exists a cycle in the subgraph induced by $\lambda$, and if so it selects such a cycle and calls the CylicSolve algorithm on it.

```
SolveBI($\mathcal{P} = (\mathcal{X}, C)$ : a bounded−increasing constraint system,
        $\rho_0$ : an initial valuation)
$\rho \leftarrow \rho_0 \vee [\![C]\!](\rho_0)$
while $[\![C]\!](\rho) \not\sqsubseteq \rho$
    $\lambda \leftarrow \emptyset$                         { $\lambda$ is a partial function from $\mathcal{X}$ to $C$ }
    repeat $|C| + 1$ times
        for each $c \in C$
            if $\rho \not\geq [\![c]\!](\rho)$
                $\rho \leftarrow \rho \vee [\![c]\!](\rho)$
                $\lambda(X) \leftarrow c$, where $X$ is the input variable of $c$
    if there exists an elementary cycle $X_0 \rightarrow \lambda(X_1) \rightarrow X_1 \cdots \lambda(X_n) \rightarrow X_0$
        construct the corresponding cyclic flattening $(\mathcal{P}', \kappa)$
        $\rho' \leftarrow \rho \circ \kappa$
        $\rho'' \leftarrow$ CyclicSolve($\mathcal{P}', \rho'$)
        $\rho \leftarrow \rho \vee \overrightarrow{\kappa}(\rho'')$
return $\rho$
```

Note that the algorithm SolveBI is an instance of the algorithm Solve where flattenings are deduced from cycles induced by the partial function $\lambda$. The following proposition 3.4 shows that this algorithm terminates.

**Proposition 3.4.** *The algorithm SolveBI returns the least solution $[\![C]\!]^*(\rho_0)$ of a bounded-increasing constraint system $\mathcal{P}$ greater than a valuation $\rho_0$. Moreover, the number of times the while loop is executed is bounded by one plus the number of constraints that are upper-saturated for $[\![C]\!]^*(\rho_0)$ but not for $\rho_0$.*

*Proof. (Sketch).* Observe that initially $\rho = \rho_0 \vee [\![C]\!](\rho_0)$. Thus, if during the execution of the algorithm $\rho(X)$ is updates by a constraint $c$ then necessary $c$ is not lower-saturated. That means if $\lambda(X)$ is defined then $c = \lambda(X)$ is not lower-saturated.

Let $\rho_0$ and $\rho_1$ be the values of $\rho$ respectively before and after the execution of the first two nested loops (line 5-9) and let $\rho_2$ be the value of $\rho$ after the execution of line 14.

Observe that if there does not exist an elementary cycle satisfying the condition given in line 11, the graph associated to $\mathcal{P}$ restricted to the edges $(X, c)$ if $c = \lambda(X)$ and the edges $(X_i, c)$ if $X_i$ is an input variable of $c$ is acyclic. This graph induces a natural partial order over the constraints $c$ of the form $c = \lambda(X)$. An enumeration $c_1, \ldots, c_m$ of this constraints compatible with the partial order provides the relation $\rho_1 \leq [\![c_1 \ldots c_m]\!](\rho_0)$. Since the loop 6-9 is executed at least $m + 1$ times, we deduce that $\rho_1$ is a solution of $\mathcal{C}$.

Lemma 3.2 shows that if $\rho_1$ is not a solution of $\mathcal{P}$ then at least one constraint is upper-saturated for $\rho_2$ but not for $\rho_0$. We deduce that the number of times the while loop is executed is bounded by one plus the number of constraints that are upper-saturated for $[\![C]\!]^*(\rho_0)$ but not for $\rho_0$. □

## 3.2 Integer Constraint Systems

A *test function* is a function $\theta_{>b}$ or $\theta_{\geq b}$ with $b \in \mathcal{Z}$ of the following form:

$$\theta_{\geq b}(x, y) = \begin{cases} y & \text{if } x \geq b \\ -\infty & \text{otherwise} \end{cases} \qquad \theta_{>b}(x, y) = \begin{cases} y & \text{if } x > b \\ -\infty & \text{otherwise} \end{cases}$$

A *test constraint* is a constraint of the form $X \geq \theta_{\sim b}(X_1, X_2)$ where $\theta_{\sim b}$ is a test function. Such a constraint $c$ is said *active* for a valuation $\rho$ if $\rho(X_1) \sim b$. Given a valuation $\rho$ such that $c$ is active, observe that $[\![c]\!](\rho)$ and $[\![c']\!](\rho)$ are equal where $c'$ is the bounded-increasing constraint $X \geq X_2$. This constraint $c'$ is called the *active form* of $c$ and denoted by $\mathsf{act}(c)$.

In the sequel, an *integer constraint* either refers to a bounded-increasing constraint or a test-constraint.

```
1  SolveInteger ($\mathcal{P} = (\mathcal{X}, C)$ : an integer constraint system)
2    $\rho \leftarrow \bot$
3    $C_t \leftarrow$ set of test constraints in $C$
```

```
4   $C' \leftarrow$ set of bounded−increasing constraints in $C$
5   while $[\![C]\!](\rho) \not\sqsubseteq \rho$
6       $\rho \leftarrow \mathsf{SolveBI}((\mathcal{X}, C'), \rho)$
7       for each $c \in C_t$
8           if $c$ is active for $\rho$
9               $C_t \leftarrow C_t \backslash \{c\}$
10              $C' \leftarrow C' \cup \{\mathsf{act}(c)\}$
11  return $\rho$
```

**Theorem 3.5.** *The algorithm* SolveInteger *computes the least solution of an integer constraint system $\mathcal{P} = (\mathcal{X}, C)$ by performing $O((|\mathcal{X}| + |C|)^3)$ integer comparisons and image computation by some bounded-increasing functions.*

*Proof.* Let us denote by $n_t$ be the number of test constraints in $C$. Observe that if during the execution of the while loop, no test constraints becomes active (line 7-10) then $\rho$ is a solution of $\mathcal{P}$ and the algorithm terminates. Thus this loop is executed at most $1+n_t$ times. Let us denote by $m_1, \ldots, m_k$ the integers such that $m_i$ is equal to the number of times the while loop of SolveBI is executed. Since after the execution there is $m_i - 1$ constraints that becomes upper-saturated, we deduce that $\sum_{i=1}^{k}(m_i - 1) \leq n$ and in particular $\sum_{i=1}^{k} m_i \leq n + k \leq 2.|C|$. Thus the algorithm SolveInteger computes the least solution of an integer constraint system $\mathcal{P} = (\mathcal{X}, C)$ by performing $O((|\mathcal{X}|+|C|)^3)$ integer comparisons and image computation by some bounded-increasing functions. □

*Remark 3.6.* We deduce that any integer constraint system is MFP-flattable.

## 4 Interval Constraints

In this section, we provide a cubic time constraint solver for intervals. Our solver is based on the usual [SW04, TG07] encoding of intervals by two integers in $\mathcal{Z}$. The main challenge is the translation of an interval constraint system with full multiplication into an integer constraint system.

An *interval* $I$ is subset of  of the form $\{x \in \ ;\ a \leq x \leq b\}$ where $a, b \in \mathcal{Z}$. We denote by $\mathcal{I}$ the complete lattice of intervals partially ordered with the inclusion relation $\sqsubseteq$. The *inverse* $-I$ of an interval $I$, the *sum* and the *multiplication* of two intervals $I_1$ and $I_2$ are defined as follows:

$$-I = \{-x;\ x \in I\} \qquad \begin{array}{l} I_1 + I_2 = \{x_1 + x_2;\ (x_1, x_2) \in I_1 \times I_2\} \\ I_1 \ . \ I_2 = \bigsqcup\{x_1.x_2;\ (x_1, x_2) \in I_1 \times I_2\} \end{array}$$

We consider interval constraints of the following forms where $I \in \mathcal{I}$:

$$X \sqsupseteq -X_1 \qquad X \sqsupseteq I \qquad X \sqsupseteq X_1 + X_2 \qquad X \sqsupseteq X_1 \sqcap I \qquad X \sqsupseteq X_1.X_2$$

Observe that we allow arbitrary multiplication between intervals, but we restrict intersection to intervals with a constant interval.

We say that an interval constraint system $\mathcal{P} = (\mathcal{X}, C)$ has the positive-multiplication property if for any constraint $c \in C$ of the form $X \sqsupseteq X_1.X_2$, the intervals $\Lambda_{\mathcal{P}}(X_1)$ and $\Lambda_{\mathcal{P}}(X_2)$ are included in . Given an interval constraint system $\mathcal{P} = (\mathcal{X}, C)$ we can effectively compute an interval constraint system $\mathcal{P}' = (\mathcal{X}', C')$ satisfying this property and such that $\mathcal{X} \subseteq \mathcal{X}'$ and $\Lambda_{\mathcal{P}}(X) = \Lambda_{\mathcal{P}'}(X)$ for any $X \in \mathcal{X}$. This constraint system $\mathcal{P}'$ is obtained from $\mathcal{P}$ by replacing the constraints $X \sqsupseteq X_1.X_2$ by the following constraints:

$$\begin{array}{ll}
X \sqsupseteq X_{1,u}.X_{2,u} & \qquad X_{1,u} \sqsupseteq X_1 \sqcap \\
X \sqsupseteq X_{1,l}.X_{2,l} & \qquad X_{2,u} \sqsupseteq X_2 \sqcap \\
X \sqsupseteq -X_{1,u}.X_{2,l} & \qquad X_{1,l} \sqsupseteq (-X_1) \sqcap \\
X \sqsupseteq -X_{1,l}.X_{2,u} & \qquad X_{2,l} \sqsupseteq (-X_2) \sqcap
\end{array}$$

Intuitively $X_{1,u}$ and $X_{2,u}$ corresponds to the positive parts of $X_1$ and $X_2$, while $X_{1,l}$ and $X_{2,l}$ corresponds to the negative parts.

Let us provide our construction for translating an interval constraint system $\mathcal{P} = (\mathcal{X}, C)$ having the positive multiplication property into an integer constraint system $\mathcal{P}' = (\mathcal{X}', C')$. Since an interval $I$ can be naturally encoded by two integers $I^-, I^+ \in \mathcal{Z}$ defined as the least upper bound of respectively $-I$ and $I$, we naturally assume that $\mathcal{X}'$ contains two integer variable $X^-$ and $X^+$ encoding each interval variable $X \in \mathcal{X}$. In order to extract from the least solution of $\mathcal{P}'$ the least solution of $\mathcal{P}$, we are looking for an integer constraint system $\mathcal{P}'$ satisfying $(\Lambda_{\mathcal{P}}(X))^- = \Lambda_{\mathcal{P}'}(X^-)$ and $(\Lambda_{\mathcal{P}}(X))^+ = \Lambda_{\mathcal{P}'}(X^+)$ for any $X \in \mathcal{X}$.

As expected, a constraint $X \sqsupseteq -X_1$ is converted into $X^+ \geq X_1^-$ and $X^- \geq X_1^+$, a constraint $X \sqsupseteq I$ into $X^+ \geq I^+$ and $X^- \geq I^-$, and a constraint $X \sqsupseteq X_1 + X_2$ into $X^- \geq X_1^- + X_2^-$ and $X^- \geq X_1^- + X_2^-$. However, a constraint $X \sqsupseteq X_1 \sqcap I$ cannot be simply translated into $X^- \geq X_1^- \wedge I^-$ and $X^+ \geq X_1^+ \wedge I^+$. In fact, these constraints may introduce imprecision when $\Lambda_{\mathcal{P}}(X) \cap I = \emptyset$. We use test functions to overcome this problem. Such a constraint is translated into the following integer constraints:

$$\begin{aligned}
X^- &\geq \theta_{\geq -I^+}(X_1^-, \theta_{\geq -I^-}(X_1^+, X_1^- \wedge I^-)) \\
X^+ &\geq \theta_{\geq -I^-}(X_1^+, \theta_{\geq -I^+}(X_1^-, X_1^+ \wedge I^+))
\end{aligned}$$

For the same reason, the constraint $X \sqsupseteq X_1.X_2$ cannot be simply converted into $X^- \geq \mathsf{mul}_-(X_1^-, X_2^-)$ and $X^+ \geq \mathsf{mul}_+(X_1^+, X_2^+)$. Instead, we consider the following constraints:

$$\begin{aligned}
X^- &\geq \theta_{>-\infty}(X_1^-, \theta_{>-\infty}(X_1^+, \theta_{>-\infty}(X_2^-, \theta_{>-\infty}(X_2^+, \mathsf{mul}_-(X_1^-, X_2^-))))) \\
X^+ &\geq \theta_{>-\infty}(X_1^+, \theta_{>-\infty}(X_1^-, \theta_{>-\infty}(X_2^+, \theta_{>-\infty}(X_2^-, \mathsf{mul}_+(X_1^+, X_2^+)))))
\end{aligned}$$

Observe in fact that $X^- \geq \mathsf{mul}_-(X_1^-, X_2^-)$ and $X^+ \geq \mathsf{mul}_+(X_1^+, X_2^+)$ are precise constraint when the intervals $I_1 = \Lambda_{\mathcal{P}}(X_1)$ and $I_2 = \Lambda_{\mathcal{P}}(X_2)$ are non empty. Since, if this condition does not hold then $I_1.I_2 = \emptyset$, the previous encoding consider this case by testing if the values of $X_1^-$, $X_1^+$, $X_2^-$, $X_2^+$ are strictly greater than $-\infty$.

Now, observe that the integer constraint system $\mathcal{P}'$ satisfies the equalities $(\Lambda_{\mathcal{P}}(X))^+ = \Lambda_{\mathcal{P}'}(X^+)$ and $(\Lambda_{\mathcal{P}}(X))^- = \Lambda_{\mathcal{P}'}(X^-)$ for any $X \in \mathcal{X}$. Thus, we have proved the following theorem.

**Theorem 4.1.** *The least solution of an interval constraint system* $\mathcal{P} = (\mathcal{X}, C)$ *with full multiplication can by computed in time* $O((|\mathcal{X}| + |C|)^3)$ *with integer manipulations performed in* $O(1)$*.*

*Remark 4.2.* We deduce that any interval constraint system is MFP-flattable.

## 5 Conclusion and Future Work

In this paper we have extended the acceleration framework from symbolic verification to the computation of MFP-solutions in data-flow analysis. Our approach leads to an efficient cubic-time algorithm for solving interval constraints with full addition and multiplication, and intersection with a constant.

As future work, it would be interesting to combine this result with strategy iteration techniques considered in [TG07] in order to obtain a polynomial time algorithm for the extension with full intersection. We also intend to investigate the application of the acceleration framework to other abstract domains.

## References

[ABS01] Annichini, A., Bouajjani, A., Sighireanu, M.: TReX: A tool for reachability analysis of complex systems. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 368–372. Springer, Heidelberg (2001)

[BFLP03] Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: FAST: Fast Acceleration of Symbolic Transition systems. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 118–121. Springer, Heidelberg (2003)

[BFLS05] Bardin, S., Finkel, A., Leroux, J., Schnoebelen, P.: Flat acceleration in symbolic model checking. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 474–488. Springer, Heidelberg (2005)

[BGWW97] Boigelot, B., Godefroid, P., Willems, B., Wolper, P.: The power of QDDs. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 172–186. Springer, Heidelberg (1997)

[BW94] Boigelot, B., Wolper, P.: Symbolic verification with periodic sets. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 55–67. Springer, Heidelberg (1994)

[CC77] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. 4th ACM Symp. Principles of Programming Languages, Los Angeles, CA, USA, pp. 238–252. ACM Press, New York (1977)

[CC92] Cousot, P., Cousot, R.: Comparing the Galois connection and widening/-narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)

[CGG$^{+}$05] Costan, A., Gaubert, S., Goubault, E., Martel, M., Putot, S.: A policy iteration algorithm for computing fixed points in static analysis of programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 462–475. Springer, Heidelberg (2005)

[CJ98] Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and Presburger arithmetic. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 268–279. Springer, Heidelberg (1998)

[FIS03] Finkel, A., Iyer, S.P., Sutre, G.: Well-abstracted transition systems: Application to FIFO automata. Information and Computation 181(1), 1–31 (2003)

[FL02] Finkel, A., Leroux, J.: How to compose Presburger-accelerations: Applications to broadcast protocols. In: Agrawal, M., Seth, A.K. (eds.) FST TCS 2002: Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 2556, pp. 145–156. Springer, Heidelberg (2002)

[Kar76] Karr, M.: Affine relationship among variables of a program. Acta Informatica 6, 133–141 (1976)

[Las] Lash homepage, `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`

[LS05] Leroux, J., Sutre, G.: Flat counter automata almost everywhere! In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 489–503. Springer, Heidelberg (2005)

[MOS04] Müller-Olm, M., Seidl, H.: A note on Karr's algorithm. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 1016–1028. Springer, Heidelberg (2004)

[SW04] Su, Z., Wagner, D.: A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 280–295. Springer, Heidelberg (2004)

[TG07] Seidl, H., Gawlitza, T.: Precise fixpoint computation through strategy iteration. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 300–315. Springer, Heidelberg (2007)

# Abstract Error Projection

Akash Lal[1], Nicholas Kidd[1], Thomas Reps[1], and Tayssir Touili[2]

[1] University of Wisconsin, Madison, Wisconsin, USA
{akash,kidd,reps}@cs.wisc.edu
[2] LIAFA, CNRS & University of Paris 7, Paris, France
touili@liafa.jussieu.fr

**Abstract.** In this paper, we extend model-checking technology with the notion of an *error projection*. Given a program abstraction, an error projection divides the program into two parts: the part outside the error projection is guaranteed to be correct, while the part inside the error projection can have bugs. Subsequent automated or manual verification effort need only be concentrated on the part inside the error projection. We present novel algorithms for computing error projections using *weighted pushdown systems* that are sound and complete for the class of Boolean programs and discuss additional applications for these algorithms.

## 1 Introduction

Software model checkers extract a model from a program using a finite abstraction of data states and then perform reachability analysis on the model. If a property violation is detected, it reports the result back to the user, usually in the form of a counterexample on a failed run, or goes on to refine its abstraction and check again. This technique has been shown to be useful both for finding program errors and for verifying certain properties of programs. It has been implemented in a number of model checkers, including SLAM [2], BLAST [11], and MAGIC [6]. Our goal is to extend the capabilities of model checkers to make maximum possible use of a given abstraction during the reachability check for helping subsequent analysis.

We accomplish this by computing *error projections* and *annotated error projections*. An error projection is the set of program nodes $N$ such that for each node $n \in N$, there exists an error path that starts from the entry point of the program and passes through $n$. By definition, an error projection describes all of the nodes that are members of paths that lead to a specified error in the model, and no more. This allows an automated program-analysis tool or human debugger to focus their efforts on only the nodes in the error projection: every node not in the error projection is correct (with respect to the property being verified). Model checkers such as SLAM can then focus their refinement effort on the part of the program inside the projection.

Annotated error projections are an extension of error projections. An *annotated error projection* adds to each node $n$ in the error projection two annotations: 1) A counterexample that passes through $n$; 2) a set of abstract stores

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 200–217, 2007.

(memory-configuration descriptors) that describes the conditions necessary at $n$ for the program to fail. The goal is to give back to the user—either an automated tool or human debugger—more of the information discovered during the model-checking process.

From a theoretical standpoint, an error projection solves a combination of forward and backward analyses. The forward analysis computes the set of program states $S_{\mathrm{fwd}}$ that are reachable from program entry; the backward analysis computes the set of states $S_{\mathrm{bck}}$ that can reach an error at certain pre-specified nodes. Under a sound abstraction of the program, each of these sets provides a strong guarantee: only the states in $S_{\mathrm{fwd}}$ can ever arise in the program, and only the states in $S_{\mathrm{bck}}$ can ever lead to error. Error projections ask the natural question of combining these guarantees to compute the set of states $S_{\mathrm{err}} = S_{\mathrm{fwd}} \cap S_{\mathrm{bck}}$ containing all states that can both arise during program execution, and lead to error. In this sense, an error projection is making maximum use of the given abstraction—by computing the smallest envelope of states that may contribute to program failure.

Computation of this intersection turns out to be non-trivial because the two sets may be infinite. In §4 and §5, we show how to compute this set efficiently and precisely for common abstractions used for model checking. We use weighted pushdown systems (WPDSs) [5,21] as the abstract model of a program, which can, among other abstractions, faithfully encode Boolean programs [22]. The techniques that we use seem to be of general interest, and apart from the application of finding error projections, we discuss additional applications in §7.

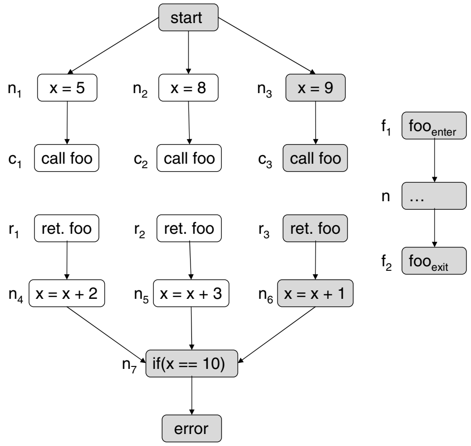
The contributions of this paper can be summarized as follows:

- We define the notions of error projection and annotated error projection. These projections divide the program into a correct and an incorrect part such that further analysis need only be carried out on the incorrect part.
- We give a novel combination of forward and backward analyses for multiprocedural programs using weighted automata and use it for computing (annotated) error projections (§4 and §5). We also show that our algorithms can be used for solving various problems in model checking (§7).
- Our experiments show that we can efficiently compute error projections (§6).

The remainder of the paper is organized as follows: §2 motivates the difficulty in computing (annotated) error projections and illustrates their utility. §3 presents the definitions of weighted pushdown systems and weighted automata. §4 and §5 give the algorithms for computing error projections and annotated error projections, respectively. §6 presents our initial experiments. §7 covers other applications of our algorithms. §8 discusses related work.

## 2 Examples

Consider the program shown in Fig. 1. Here `x` is a global unsigned integer variable, and assume that procedure `foo` does not change the value of `x`. Also assume that the program abstraction is a Boolean abstraction in which integers (only `x`



(a)

| | | |
|---|---|---|
| (1) | $\langle p, \text{start}\rangle \hookrightarrow \langle p, n_1\rangle$ | id |
| (2) | $\langle p, n_1\rangle \hookrightarrow \langle p, c_1\rangle$ | $\{(\_, 5)\}$ |
| (3) | $\langle p, c_1\rangle \hookrightarrow \langle p, f_1\ r_1\rangle$ | id |
| (4) | $\langle p, r_1\rangle \hookrightarrow \langle p, n_4\rangle$ | id |
| (5) | $\langle p, n_4\rangle \hookrightarrow \langle p, n_7\rangle$ | $\{(i, i+2)\}$ |
| (6) | $\langle p, \text{start}\rangle \hookrightarrow \langle p, n_2\rangle$ | id |
| (7) | $\langle p, n_2\rangle \hookrightarrow \langle p, c_2\rangle$ | $\{(\_, 8)\}$ |
| (8) | $\langle p, c_2\rangle \hookrightarrow \langle p, f_1\ r_2\rangle$ | id |
| (9) | $\langle p, r_2\rangle \hookrightarrow \langle p, n_5\rangle$ | id |
| (10) | $\langle p, n_5\rangle \hookrightarrow \langle p, n_7\rangle$ | $\{(i, i+3)\}$ |
| (11) | $\langle p, \text{start}\rangle \hookrightarrow \langle p, n_3\rangle$ | id |
| (12) | $\langle p, n_3\rangle \hookrightarrow \langle p, c_3\rangle$ | $\{(\_, 9)\}$ |
| (13) | $\langle p, c_3\rangle \hookrightarrow \langle p, f_1\ r_3\rangle$ | id |
| (14) | $\langle p, r_3\rangle \hookrightarrow \langle p, n_6\rangle$ | id |
| (15) | $\langle p, n_6\rangle \hookrightarrow \langle p, n_7\rangle$ | $\{(i, i+1)\}$ |
| (16) | $\langle p, n_7\rangle \hookrightarrow \langle p, \text{error}\rangle$ | $\{(10, 10)\}$ |
| (17) | $\langle p, f_1\rangle \hookrightarrow \langle p, n\rangle$ | id |
| (18) | $\langle p, n\rangle \hookrightarrow \langle p, f_2\rangle$ | id |
| (19) | $\langle p, f_2\rangle \hookrightarrow \langle p, \varepsilon\rangle$ | id |

(b)

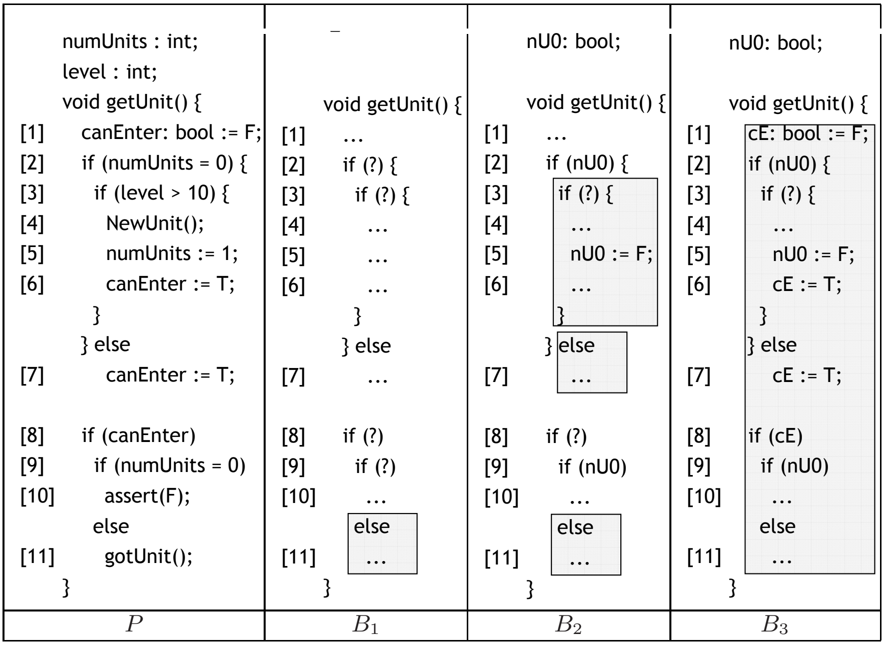
**Fig. 1.** (*a*) An example program and (*b*) its corresponding WPDS. Weights, shown in the last column, are explained in §3.

in this case) are modeled using 8 bits, i.e., the value of `x` can be between 0 and 255 with saturated arithmetic. This type of an abstraction is used by Moped [22], and happens to be a precise abstraction for this example.

The program has an error if node `error` is reached. The error projection is shaded in the figure. The paths on the left that set the value of `x` to 5 or 8 are correct paths. An error projection need not be restricted to a single trace (which would be the case if `foo` had multiple paths). An annotated error projection will additionally tell us that the value of `x` at node `n` inside `foo` has to be 9 on an error path passing through this node. Note that the value of `x` can be 5 or 8 on other paths that pass through `n`, but they do not lead to the error node.

It is non-trivial to conclude the above value of `x` for node `n`. An interprocedural forward analysis starting from `start` will show that the value of `x` is in the set $\{5, 8, 9\}$ at node `n`. A backward interprocedural analysis starting from `error` concludes that the value of `x` at `n` has to be in the set $\{7, 8, 9\}$. Intersecting the sets obtained from forward and backward analysis only gives an over-approximation of the annotated error projection values. In this case, the intersection is $\{8, 9\}$, but `x` can never be 8 on a path leading to `error`. The over-approximation occurs because, in the forward analysis, the value of `x` is 8 only when the call to `foo` occurs at call site $c_2$, but in the backward analysis the path starting at `n` with $\mathtt{x} = 8$ and leading to `error` must have had the call to `foo` from call site $c_1$.

Such a complication also occurs while computing (non-annotated) error projections: to see this, assume that the edge leading to node `n` is predicated by the condition `if(x!=9)`. Then, node `n` can be reached from `start`, and there is a path starting at `n` that leads to `error`, but both of these cannot occur together.



| *P* | $B_1$ | $B_2$ | $B_3$ |
|---|---|---|---|
| numUnits : int; | – | nU0: bool; | nU0: bool; |
| level : int; | | | |
| void getUnit() { | void getUnit() { | void getUnit() { | void getUnit() { |
| [1] canEnter: bool := F; | [1] ... | [1] ... | [1] cE: bool := F; |
| [2] if (numUnits = 0) { | [2] if (?) { | [2] if (nU0) { | [2] if (nU0) { |
| [3] if (level > 10) { | [3] if (?) { | [3] if (?) { | [3] if (?) { |
| [4] NewUnit(); | [4] ... | [4] ... | [4] ... |
| [5] numUnits := 1; | [5] ... | [5] nU0 := F; | [5] nU0 := F; |
| [6] canEnter := T; | [6] ... | [6] ... | [6] cE := T; |
| } | } | } | } |
| } else | } else | } else | } else |
| [7] canEnter := T; | [7] ... | [7] ... | [7] cE := T; |
| [8] if (canEnter) | [8] if (?) | [8] if (?) | [8] if (cE) |
| [9] if (numUnits = 0) | [9] if (?) | [9] if (nU0) | [9] if (nU0) |
| [10] assert(F); | [10] ... | [10] ... | [10] ... |
| else | else | else | else |
| [11] gotUnit(); | [11] ... | [11] ... | [11] ... |
| } | } | } | } |

**Fig. 2.** An example program *P* and its abstractions as Boolean programs. The "· · ·" represents a "skip" or a no-op. The part outside the error projection is shaded in each case.

Formally, a node is in the error projection if and only if the associated value set computed for the annotated projection is non-empty. In this sense, computing an error projection is a special case of computing the annotated version. We still discuss error projections separately because (*i*) computing them is easier, as we see later (computing annotations requires one extra trick), and (*ii*) they can very easily be cannibalized by existing model checkers such as SLAM in their abstraction-refinement phase: when an abstraction needs to be refined, only the portion inside the error projection needs to be rechecked. We illustrate this point in more detail in the next example.

Fig. 2 shows an example program and several abstractions that SLAM might produce. This example is given in [3] to illustrate the SLAM refinement process. SLAM uses predicate abstraction to create Boolean programs that abstract the original program. Boolean programs are characterized as imperative programs with procedure calls and only Boolean variables (and no heap). The Boolean programs produced as a result of predicate abstraction have one Boolean variable per predicate that tracks the value of that predicate in the program. SLAM creates successive approximations of the original program by adding more predicates. We show the utility of error projections for abstraction refinement.

First, we describe the SLAM refinement process. In Fig. 2, the property of interest is the assertion on line 10. We want to verify that line 10 is never reached (because the assertion always fails). The first abstraction $B_1$ is created without any predicates. It only reflects the control structure of $P$. Reachability analysis on $B_1$ (assuming `getUnit` is program entry) shows that the assertion is reachable. This results in a counterexample, whose subsequent analysis reveals that the predicate {`numUnits = 0`} is important. Program $B_2$ tracks that predicate using variable `nU0`. Reachability analysis on $B_2$ reveals that the assertion is still reachable. Now predicate {`canEnter = T`} is added, to produce $B_3$, which tracks the predicate's value using variable `cE`. Reachability analysis on $B_3$ reveals that the assertion is not reachable, hence it is not reachable in $P$.

The advantage of using error projections is that the whole program need not be abstracted when a new predicate is added. Analysis on $B_1$ and $B_2$ fails to prove that the whole program is correct, but error projections may reveal that at least some part of the program is correct. The parts outside the error projections (and hence correct) are shaded in the figure. Error projection on $B_1$ shows that line 11 cannot contribute to the bug, and need not be considered further. Therefore, when constructing $B_2$, we need not abstract that statement with the new predicate. Error projection on $B_2$ further reveals that lines 3 to 6 and line 7 do not contribute to the bug (the empty else branch to the conditional at line 3 still can). Thus, when $B_3$ is constructed, this part need not be abstracted with the new predicate. $B_3$, with the shaded region of $B_2$ excluded, reduces to a very simple program, resulting in reduced effort for its construction and analysis.

Annotated error projections can further reduce the analysis cost. Suppose there was some code between lines 1 and 2, possibly relevant to proving the program to be correct, that does not modify `numUnits`. After constructing $B_2$, the annotated error projection would tell us that in this region of code, `nU0` can be assumed to be *true*, because otherwise the assertion cannot be reached. This might save half of the theorem prover calls needed to abstract that region of code when using multiple predicates.

While this example did not require an interprocedural analysis, placing any piece of code inside a procedure would necessitate its use. Because Boolean programs are a common abstract model used by model checkers, we devise techniques to compute error projections precisely and efficiently on them. For this, we use weighted pushdown systems.

## 3 Preliminary Definitions

**Definition 1.** *A **pushdown system** (PDS) is a triple $\mathcal{P} = (P, \Gamma, \Delta)$ where $P$ is a finite set of states, $\Gamma$ a finite stack alphabet, and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ a finite set of rules. A **configuration** $c$ is a pair $\langle p, u\rangle$ where $p \in P$ and $u \in \Gamma^*$. The pushdown rules define a transition relation $\Rightarrow$ on configurations as follows: If $r = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta$, then $\langle p, \gamma u\rangle \Rightarrow \langle p', \gamma' u\rangle$ for all $u \in \Gamma^*$. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. For a set of configurations $C$, we define $pre^*(C) = \{c' \mid \exists c \in C : c' \Rightarrow^* c\}$ and $post^*(C) = \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$.*

Without loss of generality, we restrict PDS rules to have at most two stack symbols on the right-hand side.

A PDS is capable of encoding control flow in a program with procedures. The stack of the PDS simulates the run-time stack of the program, which stores return addresses of unfinished procedure calls, with the current program location on the top of the stack. A procedure call is modeled by a PDS rule with two stack symbols on the right-hand side: it pushes the return address on the stack before giving control to the called procedure. Procedure return is modeled by a PDS rule with no stack symbols on the right-hand side: it pops off the top of the stack and returns control to the address on the top of the stack. With such a PDS, the transition relation $\Rightarrow^*$ captures paths in the program with matched calls and returns [21,22].

Because the number of configurations of a PDS is unbounded, it is useful to use finite automata to describe certain infinite sets of configurations.

**Definition 2.** *If $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system then a $\mathcal{P}$-**automaton** is a finite automaton $(Q, \Gamma, \rightarrow, P, F)$ where $Q \supseteq P$ is a finite set of states, $\rightarrow \subseteq Q \times \Gamma \times Q$ is the transition relation, $P$ is the set of initial states, and $F$ is the set of final states. We say that a configuration $\langle p, u \rangle$ is accepted by a $\mathcal{P}$-automaton if the automaton can accept $u$ when it is started in the state $p$ (written as $p \xrightarrow{u}{}^* q$, where $q \in F$). A set of configurations is **regular** if some $\mathcal{P}$-automaton accepts it.*

If $C$ is a regular set of configurations then both $post^*(C)$ and $pre^*(C)$ are also regular sets of configurations [10,4,22]. The algorithms for computing $post^*$ and $pre^*$ take a $\mathcal{P}$-automaton $\mathcal{A}$ as input, and if $C$ is the set of configurations accepted by $\mathcal{A}$, they produce automata $\mathcal{A}_{post^*}$ and $\mathcal{A}_{pre^*}$ that accept the set of configurations $post^*(C)$ and $pre^*(C)$, respectively. In the rest of this paper, all configuration sets are regular.

A weighted pushdown system (WPDS) is a PDS augmented with a weight domain that is a bounded idempotent semiring [5,21]. The weight domain describes an abstraction with certain algebraic properties.

**Definition 3.** *A **bounded idempotent semiring** is a quintuple $(D, \oplus, \otimes, \overline{0}, \overline{1})$, where $D$ is a set whose elements are called **weights**, $\overline{0}$ and $\overline{1}$ are elements of $D$, and $\oplus$ (the combine operator) and $\otimes$ (the extend operator) are binary operators on $D$ such that*

1. *$(D, \oplus)$ is a commutative monoid with $\overline{0}$ as its neutral element, and where $\oplus$ is idempotent. $(D, \otimes)$ is a monoid with the neutral element $\overline{1}$.*
2. *$\otimes$ distributes of $\oplus$, i.e. for all $a, b, c \in D$ we have $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$.*
3. *$\overline{0}$ is an annihilator with respect to $\otimes$, i.e. for all $a \in D$, $a \otimes \overline{0} = \overline{0} = \overline{0} \otimes a$.*
4. *In the partial order $\sqsubseteq$ defined by $\forall a, b \in D, a \sqsubseteq b \iff a \oplus b = b$, there are no infinite ascending chains.*

In abstract-interpretation terminology, weights can be thought of as abstract transformers, $\otimes$ as transformer composition, and $\oplus$ as *join*. A WPDS is a PDS augmented with an abstraction (weights) and can be thought of as an abstract model of a program.

**Definition 4.** *A* ***weighted pushdown system*** *is a triple* $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ *where* $\mathcal{P} = (P, \Gamma, \Delta)$ *is a pushdown system,* $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ *is a bounded idempotent semiring and* $f : \Delta \rightarrow D$ *is a map that assigns a weight to each pushdown rule.*

Let $\sigma \in \Delta^*$ be a sequence of rules. Using $f$, we can associate a value to $\sigma$, i.e. if $\sigma = [r_1, \ldots, r_k]$, then $pval(\sigma) = f(r_1) \otimes \ldots \otimes f(r_k)$. Moreover, for any two configurations $c$ and $c'$, if $\sigma$ is a rule sequence that transitions $c$ to $c'$ then we say $c \Rightarrow^{\sigma} c'$. Reachability problems on PDSs are generalized to WPDSs as follows:

**Definition 5.** *Let* $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ *be a WPDS, where* $\mathcal{P} = (P, \Gamma, \Delta)$*, and let* $S, T \subseteq P \times \Gamma^*$ *be regular sets of configurations. Then the* ***join-over-all-paths*** *value* $JOP(S, T)$ *is defined as* $\bigoplus\{pval(\sigma) \mid s \Rightarrow^{\sigma} t, s \in S, t \in T\}$.

A PDS is a WPDS with the Boolean weight domain $(\{\overline{1}, \overline{0}\}, \oplus, \otimes, \overline{0}, \overline{1})$ and $f(r) = \overline{1}$ for all rules $r \in \Delta$. (JOP$(S, T) = \overline{1}$ iff a configuration in $S$ can reach a configuration in $T$.) In §5 we use the weight domain of all binary relations on a finite set:

**Definition 6.** *Let* $V$ *be a finite set. A* ***relational weight domain*** *on* $V$ *is defined as the semiring* $(D, \oplus, \otimes, \overline{0}, \overline{1})$ *where* $D = \mathcal{P}(V \times V)$ *is the set of all binary relations on* $V$*,* $\oplus$ *is union,* $\otimes$ *is relational composition,* $\overline{0}$ *is the empty set, and* $\overline{1}$ *is the identity relation.*

Such domains are useful for describing finite abstractions, e.g., predicate abstraction, abstraction of Boolean programs, and finite-state safety properties (a short discussion can be found in [16]). In predicate abstraction, $v \in V$ would be a fixed valuation of the predicates, which in turn represents all memory configurations in which that valuation holds. Weights are transformations on these states that represent the abstract effect of executing a program statement. They can usually be represented succinctly using BDDs. (This is the essence of Schwoon's Moped system [22]).

For the program shown in Fig. 1 and an 8-bit integer abstraction (explained in §2), the WPDS uses a relational weight domain over the set $V = \{0, 1, \cdots, 255\}$. The weight $\{(\_, 5)\}$ is shorthand for the set $\{(i, 5) \mid i \in V\}$; $\{(i, i+1)\}$ stands for $\{(i, i+1) \mid i \in V\}$ (with saturated arithmetic); and *id* stands for the identity relation on $V$.

**Solving for the JOP value in WPDSs.** There are two algorithms for finding the JOP value, called *poststar* and *prestar*, based on forward and backward reachability, respectively [21]. These algorithms operate on *weighted automata* defined as follows.

**Definition 7.** *Given a WPDS* $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$*, a* $\mathcal{W}$***-automaton*** $\mathcal{A}$ *is a* $\mathcal{P}$*-automaton, where each transition in the automaton is labeled with a weight. The weight of a path in the automaton is obtained by taking an extend of the weights on the transitions in the path in either a forward or backward direction. The automaton is said to accept a configuration* $c = \langle p, u \rangle$ *with weight* $w$*, written as* $\mathcal{A}(c)$*, if* $w$ *is the combine of weights of all accepting paths for* $u$ *starting from state*

*$p$ in the automaton. We call the automaton a* ***backward $\mathcal{W}$-automaton*** *if the weight of the path is read backwards and a* ***forward $\mathcal{W}$-automaton*** *otherwise.*

For simplicity, we call a $\mathcal{W}$-automaton a weighted automaton. The poststar algorithm takes a backward weighted automaton $\mathcal{A}$ as input and produces another backward weighted automaton $\mathit{poststar}(\mathcal{A})$, such that $\mathit{poststar}(\mathcal{A})(c) = \bigoplus\{\mathcal{A}(c') \otimes \mathit{pval}(\sigma) \mid c' \Rightarrow^{\sigma} c\}$. Similarly, the prestar algorithm takes a forward weighted automaton $\mathcal{A}$ and produces $\mathit{prestar}(\mathcal{A})$ such that $\mathit{prestar}(\mathcal{A})(c) = \bigoplus\{\mathit{pval}(\sigma) \otimes \mathcal{A}(c') \mid c \Rightarrow^{\sigma} c'\}$.

We briefly describe how the prestar algorithm works. The interested reader is referred to [21] for more details, and an efficient implementation of the algorithm. The algorithm takes a weighted automaton $\mathcal{A}$ as input, and adds weighted transitions to it until no more can be added. The addition of transitions is based on the following rule: for a WPDS rule $r = \langle p, \gamma\rangle \hookrightarrow \langle q, \gamma_1\gamma_2 \cdots \gamma_n\rangle$ with weight $f(r)$ and transitions $(q, \gamma_1, q_1), (q_1, \gamma_2, q_2), \cdots, (q_{n-1}, \gamma_n, q_n)$ with weights $w_1, w_2, \cdots, w_n$, add the transition $(p, \gamma, q_n)$ to $\mathcal{A}$ with weight $w = f(r) \otimes w_1 \otimes \cdots \otimes w_n$. If this transition already exists with weight $w'$, change the weight to $w \oplus w'$. This algorithm is based on the intuition that if the automaton accepts configurations $c$ and $c'$ with weights $w$ and $w'$, respectively, and rule $r$ allows the transition $c' \Rightarrow c$, then the automaton is changed to accept $c'$ with weight $w' \oplus (f(r) \otimes w)$. Termination follows from the fact that the number of states of the automaton does not increase (hence, the number of transitions is bounded), and that the weight domain satisfies the ascending-chain condition.

An important algorithm for reading out weights from weighted automata is called *path_summary* defined as follows: $\mathit{path\_summary}(\mathcal{A}) = \oplus\{\mathcal{A}(c) \mid c \in P \times \Gamma^*\}$. We briefly outline this algorithm for a forward weighted automaton. It is based on a standard fixpoint-finding algorithm. It associates a weight $l(q)$ to each state $q$ of $\mathcal{A}$: Initialize the weight of each non-initial state in $\mathcal{A}$ to $\overline{0}$ and each initial state to $\overline{1}$; add each initial state to a worklist. Next, repeatedly remove a state, say $q$, from the worklist and propagate its weight forwards: i.e., if there is a transition $(q, \gamma, q')$ with weight $w$, then update the weight of state $q'$ as $l(q') := l(q') \oplus (l(q) \otimes w)$; if the weight on $q'$ changes, then add it to the worklist. This is repeated until the worklist is empty. Then $\mathit{path\_summary}(\mathcal{A})$ is the combine of $l(q)$ for each final state $q$.

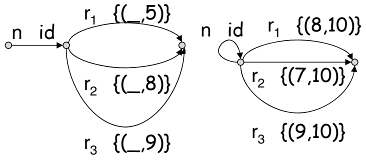


**Fig. 3.** Parts of the *poststar* and *prestar* automaton, respectively

Using *path_summary*, we can calculate $\mathcal{A}(C) = \bigoplus\{\mathcal{A}(c) \mid c \in C\}$ as follows: Let $\mathcal{A}_C$ be an (unweighted) automaton that accepts $C$. Intersect $\mathcal{A}$ and $\mathcal{A}_C$ to obtain a weighted automaton $\mathcal{A}'$.[1] Then it is easy to see that $\mathcal{A}(C) = \mathit{path\_summary}(\mathcal{A}')$. Using this, we can solve for JOP. Let $\mathcal{A}_S$ and $\mathcal{A}_T$ be (unweighted) automata that accept the sets $S$ and $T$, respectively. Then $\mathrm{JOP}(S, T)$

[1] Intersection of a weighted automaton with an unweighted one is carried out the same way as for two unweighted automata, except that the weights of the weighted automaton are copied over to the resultant automaton.

= $\mathit{poststar}(\mathcal{A}_S)(T)$ = $\mathit{prestar}(\mathcal{A}_T)(S)$. For the program shown in Fig. 1, parts of the automata produced by $\mathit{poststar}(\{\texttt{start}\})$ and $\mathit{prestar}(\texttt{error}\ \Gamma^*)$ are shown in Fig. 3 (only the part important for node n is shown).[2] Using these, we get $\mathrm{JOP}(\{\texttt{start}\},\texttt{n}\ \Gamma^*) = \{(\_,5),(\_,8),(\_,9)\}$ and $\mathrm{JOP}(\texttt{n}\ \Gamma^*,\texttt{error}\ \Gamma^*) = \{(7,10),(8,10),(9,10)\}$. Here, $(\gamma\ \Gamma^*)$ stands for the set $\{\gamma\ c \mid c \in \Gamma^*\}$.

# 4 Computing an Error Projection

Let us now define an error projection using WPDSs as our model of programs. Usually, a WPDS created from a program has a single PDS state. Even when this is not the case, the states can be pushed inside the weights to get a single-state WPDS. We use this to simplify the discussion: PDS configurations are just represented as stacks ($\Gamma^*$).

Also, we concern ourselves with assertion checking. We assume that we are given a target set of control configurations $T$ such that the program model exhibits an error only if it can reach a configuration in that set. One way of accomplishing this is to convert every assertion of the form "$\texttt{assert(}\mathcal{E}\texttt{)}$" into a condition "$\texttt{if(!}\mathcal{E}\texttt{) then goto error}$" (assuming $!\mathcal{E}$ is expressible under the current abstraction), and instantiate $T$ to be the set of configurations ($\texttt{error}\ \Gamma^*$). We also assume that the weight abstraction has been constructed such that a path $\sigma$ in the PDS is *infeasible* if and only if its weight $\mathit{pval}(\sigma)$ is $\overline{0}$. Therefore, under this model, the program has an error only when it can reach a configuration in $T$ with a path of non-$\overline{0}$ weight.

**Definition 8.** *Given $S$, the set of starting configurations of the program, and a target set of configurations $T$, a program node $\gamma \in \Gamma$ is in the **error projection** $EP(S,T)$ if and only if there exists a path $\sigma = \sigma_1\sigma_2$ such that $\mathit{pval}(\sigma) \neq \overline{0}$ and $s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t$ for some $s \in S, c \in \gamma\Gamma^*, t \in T$.*

We calculate the error projection by computing a constrained form of the join-over-all-paths value, which we call a weighted chopping query.

**Definition 9.** *Given regular sets of configurations $S$ (source), $T$ (target), and $C$ (chop); a **weighted chopping query** is to compute the following weight:*

$$\mathit{WC}(S,C,T) = \bigoplus\{v(\sigma_1\sigma_2) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, s \in S, c \in C, t \in T\}$$

It is easy to see that $\gamma \in \mathrm{EP}(S,T)$ if and only if $\mathrm{WC}(S,\gamma\ \Gamma^*,T) \neq \overline{0}$. We now show how to solve these queries. First, note that $\mathrm{WC}(S,C,T) \neq \mathrm{JOP}(S,C) \otimes \mathrm{JOP}(C,T)$. For example, in Fig. 1, if foo was not called from $\texttt{c}_3$, and $S = \{\texttt{start}\}, T = (\texttt{error}\ \Gamma^*), C = (\texttt{n}\ \Gamma^*)$ then $\mathrm{JOP}(S,C) = \{(\_,5),(\_,8)\}$ and $\mathrm{JOP}(C,T) = \{(7,10),(8,10)\}$, and their extend is non-empty, whereas $\mathrm{WC}(S,C,T) = \emptyset$. This is exactly the problem mentioned in §2.

[2] Intuitively, for the *poststar* automaton, the weight on a transition labeled with $\gamma$ is the net transformer to go from the entry of the procedure containing $\gamma$ to $\gamma$. For the *prestar* automaton, it is the transformer to go from $\gamma$ to the exit of the procedure.

A first attempt at solving weighted chopping is to use the identity $\mathrm{WC}(S, C, T) = \bigoplus\{\mathrm{JOP}(S, c) \otimes \mathrm{JOP}(c, T) \mid c \in C\}$. However, this only works when $C$ is a finite set of configurations, which is not the case if we want to compute an error projection. We can solve this problem using the automata-theoretic constructions described in the previous section. Let $\mathcal{A}_S$ be an unweighted automaton that represents the set $S$, and similarly for $\mathcal{A}_C$ and $\mathcal{A}_T$. The following two algorithms, given in different columns, are valid ways of solving a weighted chopping query.

| | |
|---|---|
| 1. $\mathcal{A}_1 = \mathit{poststar}(\mathcal{A}_S)$ | 1. $\mathcal{A}_1 = \mathit{prestar}(\mathcal{A}_T)$ |
| 2. $\mathcal{A}_2 = (\mathcal{A}_1 \cap \mathcal{A}_C)$ | 2. $\mathcal{A}_2 = (\mathcal{A}_1 \cap \mathcal{A}_C)$ |
| 3. $\mathcal{A}_3 = \mathit{poststar}(\mathcal{A}_2)$ | 3. $\mathcal{A}_3 = \mathit{prestar}(\mathcal{A}_2)$ |
| 4. $\mathcal{A}_4 = \mathcal{A}_3 \cap \mathcal{A}_T$ | 4. $\mathcal{A}_4 = \mathcal{A}_3 \cap \mathcal{A}_S$ |
| 5. $\mathrm{WC}(S, C, T) = \mathit{path\_summary}(\mathcal{A}_4)$ | 5. $\mathrm{WC}(S, C, T) = \mathit{path\_summary}(\mathcal{A}_4)$ |

The running time is only proportional to the size of $\mathcal{A}_C$, not the size of the language accepted by it. A proof of correctness can be found in [15].

An error projection is computed by solving a separate weighted chopping query for each node $\gamma$ in the program. This means that the source set $S$ and the target set $T$ remain fixed, but the chop set $C$ keeps changing. Unfortunately, the two algorithms given above have a major shortcoming: only their first steps can be carried over from one chopping query to the next; the rest of the steps have to be recomputed for each node $\gamma$. As shown in §6, this approach is very slow, and the algorithm discussed next is about 3 orders of magnitude faster.

To derive a better algorithm for weighted chopping that is more suited for computing error projections, let us first look at the unweighted case (i.e., the weighted case where the weight domain just contains the weights $\overline{0}$ and $\overline{1}$). Then $\mathrm{WC}(S, C, T) = \overline{1}$ if and only if $(\mathit{post}^*(S) \cap \mathit{pre}^*(T)) \cap C \neq \emptyset$. This procedure just requires a single intersection operation for different chop sets. Computation of both $\mathit{post}^*(S)$ and $\mathit{pre}^*(T)$ have to be done just once. We generalize this approach to the weighted case.

First, we need to define what we mean by intersecting weighted automata. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two weighted automata. Define their *intersection* $\mathcal{A}_1 \lhd \mathcal{A}_2$ to be a function from configurations to weights, which we later compute in the form of a weighted automaton, such that $(\mathcal{A}_1 \lhd \mathcal{A}_2)(c) = \mathcal{A}_1(c) \otimes \mathcal{A}_2(c)$.[3] Define $(\mathcal{A}_1 \lhd \mathcal{A}_2)(C) = \bigoplus\{(\mathcal{A}_1 \lhd \mathcal{A}_2)(c) \mid c \in C\}$, as before. Based on this definition, if $\mathcal{A}_{post^*} = \mathit{poststar}(\mathcal{A}_S)$ and $\mathcal{A}_{pre^*} = \mathit{prestar}(\mathcal{A}_T)$, then $\mathrm{WC}(S, C, T) = (\mathcal{A}_{post^*} \lhd \mathcal{A}_{pre^*})(C)$.

Let us give some intuition into why intersecting weighted automata is hard. For $\mathcal{A}_1$ and $\mathcal{A}_2$ as above, the intersection is defined to read off the weight from $\mathcal{A}_1$ first and then extend it with the weight from $\mathcal{A}_2$. A naive approach would be to construct a weighted automaton $\mathcal{A}_{12}$ as the concatenation of $\mathcal{A}_1$ and $\mathcal{A}_2$ (with epsilon transitions from the final states of $\mathcal{A}_1$ to the initial states of $\mathcal{A}_2$) and let

[3] Note that the operator $\lhd$ is not commutative is general, but we still call it *intersection* because the construction of $\mathcal{A}_1 \lhd \mathcal{A}_2$ resembles the one for intersection of unweighted automata.

$(\mathcal{A}_1 \lhd \mathcal{A}_2)(c) = \mathcal{A}_{12}(c\ c)$. However, computing $(\mathcal{A}_1 \lhd \mathcal{A}_2)(C)$ for a regular set $C$ requires computing join-over-all-paths in $\mathcal{A}_{12}$ over the set of paths that accept the language $\{(c\ c) \mid c \in C\}$ because the *same* path (i.e., $c$) must be followed in both $\mathcal{A}_1$ and $\mathcal{A}_2$. This language is neither regular nor context-free, and we do not know of any method that computes join-over-all-paths over a non-context-free set of paths.

The trick here is to recognize that weighted automata have a direction in which weights are read off. We need to intersect $\mathcal{A}_{post^*}$ with $\mathcal{A}_{pre^*}$, where $\mathcal{A}_{post^*}$ is a backward automaton and $\mathcal{A}_{pre^*}$ is a forward automaton. If we concatenate these together but reverse the second one (reverse all transitions and switch initial and final states), then we get a purely backward weighted automaton and we only need to solve for join-over-all-paths over the language $\{(c\ c^R) \mid c \in C\}$ where $c^R$ is $c$ written in the reverse order. This language can be defined using a linear context-free grammar with production rules of the form "$X \rightarrow \gamma Y \gamma$", where $X$ and $Y$ are non-terminals. The following section uses this intuition to derive an algorithm for intersecting two weighted automata.

**Intersecting Weighted Automata.** Let $\mathcal{A}_b = (Q_b, \Gamma, \rightarrow_b, P, F_b)$ be a backward weighted automaton and $\mathcal{A}_f = (Q_f, \Gamma, \rightarrow_f, P, F_f)$ be a forward weighted automaton. We proceed with the standard automata-intersection algorithm: Construct a new automaton $\mathcal{A}_{bf} = (Q_b \times Q_f, \Gamma, \rightarrow, P, F_b \times F_f)$, where we identify the state $(p,p), p \in P$ with $p$, i.e., the $P$-states of $\mathcal{A}_{bf}$ are states of the form $(p,p), p \in P$. The transitions of this automaton are computed by matching on stack symbols. If $t_b = (q_1, \gamma, q_2)$ is a transition in $\mathcal{A}_b$ with weight $w_b$ and $t_f = (q_3, \gamma, q_4)$ is a transition in $\mathcal{A}_f$ with weight $w_f$, then add transition $t_{bf} = ((q_1,q_3), \gamma, (q_2,q_4))$ to $\mathcal{A}_{bf}$ with weight $\lambda z.(w_b \otimes z \otimes w_f)$. We call this type of weight a *functional weight* and use the capital letter $W$ (possibly subscripted) to distinguish them from normal weights. Functional weights are special functions on weights: given a weight $w$ and a functional weight $W = \lambda z.(w_1 \otimes z \otimes w_2)$, $W(w) = (w_1 \otimes w \otimes w_2)$. The automaton $\mathcal{A}_{bf}$ is called a *functional automaton.*

We define extend on functional weights as reversed function composition. That is, if $W_1 = \lambda z.(w_1 \otimes z \otimes w_2)$ and $W_2 = \lambda z.(w_3 \otimes z \otimes w_4)$, then $W_1 \otimes W_2 = W_2 \circ W_1 = \lambda z.((w_3 \otimes w_1) \otimes z \otimes (w_2 \otimes w_4))$, and is thus also a functional weight. However, the combine operator, defined as $W_1 \oplus W_2 = \lambda z.(W_1(z) \oplus W_2(z))$, does not preserve the form of functional weights. Hence, functional weights do not form a semiring. We now show that this is not a handicap, and we can still compute $\mathcal{A}_b \lhd \mathcal{A}_f$ as required.

Because $\mathcal{A}_{bf}$ is a product automaton, every path in it of the form $(q_1,q_2) \xrightarrow{c}{}^* (q_3,q_4)$ is in one-to-one correspondence with paths $q_1 \xrightarrow{c}{}^* q_3$ in $\mathcal{A}_b$ and $q_2 \xrightarrow{c}{}^* q_4$ in $\mathcal{A}_f$. Using this fact, we get that the weight of a path in $\mathcal{A}_{bf}$ will be a function of the form $\lambda z.(w_b \otimes z \otimes w_f)$, where $w_b$ and $w_f$ are the weights of the corresponding paths in $\mathcal{A}_b$ and $\mathcal{A}_f$, respectively. In this sense, $\mathcal{A}_{bf}$ is constructed based on the intuition given in the previous section: the functional weights resemble grammar productions "$X \rightarrow \gamma Y \gamma$" for the language $\{(c\ c^R)\}$ with weights replacing the two occurrences of $\gamma$, and their composition resembles the derivation of a string in the language. (Note that in "$X \rightarrow \gamma Y \gamma$", the first $\gamma$ is a letter in $c$, whereas the second $\gamma$ is a letter in $c^R$. In general, the letters will be given different weights in $\mathcal{A}_b$ and $\mathcal{A}_f$.)

Formally, for a configuration $c$ and a weighted automaton $\mathcal{A}$, define the predicate $accpath(\mathcal{A}, c, w)$ to be true if there is an accepting path in $\mathcal{A}$ for $c$ that has weight $w$, and false otherwise (note that we only need the extend operation to compute the weight of a path). Similarly, $accpath(\mathcal{A}, C, w)$ is true iff $accpath(\mathcal{A}, c, w)$ is true for some $c \in C$. Then we have:

$$\begin{aligned}(\mathcal{A}_b \lhd \mathcal{A}_f)(c) &= \mathcal{A}_b(c) \otimes \mathcal{A}_f(c)\\ &= \bigoplus\{w_b \otimes w_f \mid accpath(\mathcal{A}_b, c, w_b), accpath(\mathcal{A}_f, c, w_f)\}\\ &= \bigoplus\{w_b \otimes w_f \mid accpath(\mathcal{A}_{bf}, c, \lambda z.(w_b \otimes z \otimes w_f))\}\\ &= \bigoplus\{\lambda z.(w_b \otimes z \otimes w_f)(\overline{1}) \mid accpath(\mathcal{A}_{bf}, c, \lambda z.(w_b \otimes z \otimes w_f))\}\\ &= \bigoplus\{W(\overline{1}) \mid accpath(\mathcal{A}_{bf}, c, W)\}\end{aligned}$$

Similarly, we have $(\mathcal{A}_b \lhd \mathcal{A}_f)(C) = \bigoplus\{W(\overline{1}) \mid accpath(\mathcal{A}_{bf}, C, W)\} = \bigoplus\{W(\overline{1}) \mid accpath(\mathcal{A}_{bf} \cap \mathcal{A}_C, \Gamma^*, W)\}$, where $\mathcal{A}_C$ is an unweighted automaton that accepts the set $C$, and this can be obtained using a procedure similar to *path_summary*. The advantage of the way we have defined $\mathcal{A}_{bf}$ is that we can intersect it with $\mathcal{A}_C$ (via ordinary intersection) and then run *path_summary* over it, as we show next.

Functional weights distribute over (ordinary) weights, i.e., $W(w_1 \oplus w_2) = W(w_1) \oplus W(w_2)$. Thus, *path_summary*$(\mathcal{A}_{bf})$ can be obtained merely by solving an intraprocedural join-over-all-paths over distributive transformers starting with the weight $\overline{1}$, which is completely standard: Initialize $l(q) = \overline{1}$ for initial states, and set $l(q) = \overline{0}$ for other states. Then, until a fixpoint is reached, for a transition $(q, \gamma, q')$ with weight $W$, update the weight on state $q'$ by $l(q') := l(q') \oplus W(l(q))$. Then *path_summary*$(\mathcal{A}_{bf})$ is the combine of the weights on the final states. Termination is guaranteed because we still have weights associated with states, and functional weights are monotonic. Because of the properties satisfied by $\mathcal{A}_{bf}$, we use $\mathcal{A}_{bf}$ as a representation for $(\mathcal{A}_b \lhd \mathcal{A}_f)$.

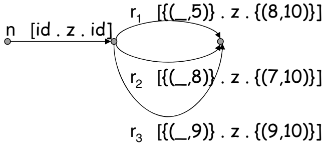

n [id . z . id]
r1 [{(_,5)} . z . {(8,10)}]
r2 [{(_,8)} . z . {(7,10)}]
r3 [{(_,9)} . z . {(9,10)}]

**Fig. 4.** Functional automaton obtained after intersecting the automata of Fig. 3

This allows us to solve $\mathrm{WC}(S, C, T) = (\mathcal{A}_{post^*} \lhd \mathcal{A}_{pre^*})(C)$. That is, after a preparation step to create $(\mathcal{A}_{post^*} \lhd \mathcal{A}_{pre^*})$, one can solve $\mathrm{WC}(S, C, T)$ for different chop sets $C$ just using intersection with $\mathcal{A}_C$ followed by *path_summary*, as shown above. Fig. 4 shows an example. For short, the weight $\lambda z.(w_1 \otimes z \otimes w_2)$ is denoted by $[w_1.z.w_2]$. Note how the weights get appropriately paired for different call sites.

It should be noted that this technique applies only to the intersection of a forward weighted automaton with a backward one, because in this case we are able to get around the problem of computing join-over-all-paths over a non-context-free set of paths. We are not aware of any algorithms for intersecting two forward or two backward automata; those problems remain open.

## 5 Computing an Annotated Error Projection

An annotated error projection adds more information to an error projection by associating each node in the error projection with (*i*) at least one counterexample that goes through that node and (*ii*) the set of *abstract stores* (or memory descriptors) that may arise on a path doomed to fail in the future. Due to space constraints, we do not discuss the first part here. It can be found in [15].

For defining and computing the abstract stores for nodes in an error projection, we restrict ourselves to relational abstractions over a finite set. We can only compute the precise set of abstract stores under this assumption. In other cases, we can only approximate the desired set of abstract stores (the approximation algorithms are given in [15]). Note that the value of $\mathrm{WC}(S, C, T)$ does not say anything about the required set of abstract stores at $C$: for Fig. 1, $\mathrm{WC}(S, \mathtt{n}\ \Gamma^*, T) = \{(\_, 10)\}$ but the required abstract store at $\mathtt{n}$ is $\{9\}$.

Let $V$ be a finite set of abstract stores and $(D, \oplus, \otimes, \overline{0}, \overline{1})$ the relational weight domain on $V$, as defined in Defn. 6. For weights $w, w_1, w_2 \in D$, define $\mathtt{Rng}(w)$ to be the range of $w$, $\mathtt{Dom}(w)$ to be the domain of $w$ and $\mathtt{Com}(w_1, w_2) = \mathtt{Rng}(w_1) \cap \mathtt{Dom}(w_2)$. For a node $\gamma \in \mathrm{EP}(S, T)$, we compute the following subset of $V$: $V_\gamma = \{v \in \mathtt{Com}(\mathit{pval}(\sigma_1), \mathit{pval}(\sigma_2)) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, s \in S, c \in \gamma\Gamma^*, t \in T\}$. If $v \in V_\gamma$, then there must be a path in the program model that leads to an error such that the abstract store $v$ arises at node $\gamma$.

**An Explicit Algorithm.** First, we show how to check for membership in the set $V_\gamma$. Conceptually, we place a bottleneck at node $\gamma$, using a special weight, to see if there is a feasible path that can pass through the bottleneck at $\gamma$ with abstract store $v$, and then continue on to the error configuration. Let $w_v = \{(v, v)\}$. Note that $v \in \mathtt{Com}(w_1, w_2)$ iff $w_1 \otimes w_v \otimes w_2 \neq \overline{0}$. Let $\mathcal{A}_{post^*} = \mathit{poststar}(\mathcal{A}_S), \mathcal{A}_{pre^*} = \mathit{prestar}(\mathcal{A}_T)$ and $\mathcal{A}_\lhd$ be their intersection. Then $v \in V_\gamma$ iff there is a configuration $c \in \gamma\Gamma^*$ such that $\mathrm{JOP}(S, c) \otimes w_v \otimes \mathrm{JOP}(c, T) \neq \overline{0}$ or, equivalently, $\mathcal{A}_{post^*}(c) \otimes w_v \otimes \mathcal{A}_{pre^*}(c) \neq \overline{0}$. To check this, we use the functional automaton $\mathcal{A}_\lhd$ again. It is not hard to check that the following holds for any weight $w$:

$$\mathcal{A}_{post^*}(c) \otimes w \otimes \mathcal{A}_{pre^*}(c) = \bigoplus\{W(w) \mid \mathit{accpath}(\mathcal{A}_\lhd, c, W)\}$$

Then $v \in V_\gamma$ iff $\bigoplus\{W(w_v) \mid \mathit{accpath}(\mathcal{A}_\lhd, \gamma\Gamma^*, W)\} \neq \overline{0}$. This is, again, computable using *path_summary*: Intersect $\mathcal{A}_\lhd$ with an unweighted automaton accepting $\gamma\Gamma^*$, then run *path_summary* but initialize the weight on initial states with $w_v$ instead of $\overline{1}$.

This gives us an algorithm for computing $V_\gamma$, but its running time would be proportional to $|V|$, which might be very large. In the case of predicate abstraction, $|V|$ is exponential in the number of predicates, but the weights (transformers) can be efficiently encoded using BDDs. For example, the identity transformer on $V$ can be encoded with a BDD of size $\log |V|$. To avoid losing the advantages of using BDDs, we now present a symbolic algorithm.

**A Symbolic Algorithm.** Let $Y = \{y_v \mid v \in V\}$ be a set of variables. We switch our weight domain from being $V \times V$ to $V \times Y \times V$. We write weights in the new domain with superscript $e$. Intuitively, the triple $(v_1, y, v_2)$

denotes the transformation of $v_1$ to $v_2$ provided the variable $y$ is "true". Combine is still defined to be union and extend is defined as follows: $w_1^e \otimes w_2^e = \{(v_1, y, v_2) \mid (v_1, y, v_3) \in w_1^e, (v_3, y, v_2) \in w_2^e\}$. Also, $\overline{1}^e = \{(v, y, v) \mid v \in V, y \in Y\}$ and $\overline{0}^e = \emptyset$. Define a symbolic identity $\mathrm{id}_s^e$ as $\{(v, y_v, v) \mid v \in V\}$. Let $\mathtt{Var}(w^e) = \{v \mid (v_1, y_v, v_2) \in w^e \text{ for some } v_1, v_2 \in V\}$, i.e., the set of values whose corresponding variable appears in $w^e$. Given a weight in $V \times V$, define $\mathtt{ext}(w) = \{(v_1, y, v_2) \mid (v_1, v_2) \in w, y \in Y\}$, i.e., all variables are added to the middle dimension. We will use the middle dimension to remember the "history" when composition is performed: for weights $w_1, w_2 \in V \times V$, it is easy to prove that $\mathtt{Com}(w_1, w_2) = \mathtt{Var}(\mathtt{ext}(w_1) \otimes \mathrm{id}_s^e \otimes \mathtt{ext}(w_2))$. Therefore, $V_\gamma = \mathtt{Var}(w_\gamma^e)$ where, $w_\gamma^e = \bigoplus\{\mathtt{ext}(\mathit{pval}(\sigma_1)) \otimes \mathit{id}_s^e \otimes \mathtt{ext}(\mathit{pval}(\sigma_2)) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, s \in S, c \in \gamma\Gamma^*, t \in T\}$. This weight is computed by replacing all weights $w$ in the functional automaton with $\mathtt{ext}(w)$ and running *path_summary* over paths accepting $\gamma\Gamma^*$, and initializing initial states with weight $\mathrm{id}_s^e$. The advantages of this algorithm are: the weight $\mathtt{ext}(w)$ can be represented using the same-sized BDD as the one for $w$ (the middle dimension is "don't-care"); and the weight $\mathrm{id}_s^e$ can be represented using a BDD of size $O(\log |V|)$.

For our example, the weight $w_{\mathtt{n}}^e$ read off from the functional automaton shown in Fig. 4 is $\{(\_, y_9, 10)\}$, which gives us $V_{\mathtt{n}} = \{9\}$, as desired.

## 6 Experiments

We added the error-projection algorithm to Moped [22], a program-analysis tool that encodes Boolean programs as WPDSs and answers reachability queries on them for checking assertions. The Boolean programs may be obtained after performing predicate abstraction or from integer programs with a limited number of bits to represent bounded integers. Although it uses a finite abstraction, the use of weights to encode abstract transformers as BDDs is crucial for its scalability. Because we can compute an error projection using just extend and combine, we take full advantage of the BDD encoding.

We measured the time needed to solve $\mathrm{WC}(S, n\Gamma^*, T)$ for all program nodes $n$ using the algorithms from §4: one that uses functional automata and one based on running two *prestar* queries (called the double-*pre** method below). Although we report the size of the error projection, we could not validate how useful it was because only the model (and not the source code) was available to us.

The results are shown in Tab. 1. The table can be read as follows: the first five columns give the program names, the number of nodes (or basic blocks) in the program, error-projection size relative to program size, and times to compute $\mathit{post}^*(S)$ and $\mathit{pre}^*(T)$, respectively. The next two columns give the running time for solving $\mathrm{WC}(S, n\Gamma^*, T)$ for all nodes $n$ using functionals and using double-*pre**, after the initial computation of $\mathit{post}^*(S)$ and $\mathit{pre}^*(T)$ was completed. Because the double-*pre** method is so slow, we did not run these examples to completion; instead, we report the time for solving the weighted chop query for only 1% of the blocks and multiply the resulting number by 100. The last two columns compare the running time for using functionals (column six) against the

**Table 1.** MOPED results: The WPDSs are models of Boolean programs provided by S. Schwoon. $S$ is the entry point of the program, and $T$ is the error configuration set. An error projection of size 0% means that the program is correct.

| | | | | | WC($S, n\Gamma^*, T$) | | Functional vs. | |
|---|---|---|---|---|---|---|---|---|
| Prog | Nodes | Error Proj. | $post^*(S)$ | $pre^*(T)$ | Functional | Double $pre^*$ | Reach | Double $pre^*$ |
| iscsiprt16 | 4884 | 0% | 79 | 1.8 | 3.5 | 5800 | 0.04 | 1657 |
| pnpmem2 | 4813 | 0% | 7 | 4.1 | 8.8 | 16000 | 0.79 | 1818 |
| iscsiprt10 | 4824 | 46% | 0.28 | 0.36 | 1.6 | 1200 | 2.5 | 750 |
| pnpmem1 | 4804 | 65% | 7.2 | 4.5 | 9.2 | 17000 | 0.79 | 1848 |
| iscsi1 | 6358 | 84% | 53 | 110 | 140 | 750000 | 0.88 | 5357 |
| bugs5 | 36972 | 99% | 13 | 2 | 170 | 85000 | 11.3 | 500 |

time taken to compute $post^*(S) + pre^*(T)$; and the time taken by the double-$pre^*$ method. All running times are in seconds. The experiments were run on a 3GHz P4 machine with 2GB RAM.

**Discussion.** As can be seen from the table, using functionals is about three orders of magnitude faster than using the double-$pre^*$ method. Also, as shown in column eight, computation of the error projection compares fairly well with running a single forward or backward analysis (at least for the smaller programs). To some extent, this implies that error-projection computation can be incorporated into model checkers without adding significant overhead.

The sizes of the error projections indicate that they might be useful in model checkers. Simple slicing, which only deals with the control structure of the program (and no weights) produced more than 99% of the program in each case, even when the program was correct.

The result for the last program `bugs5`, however, does not seem as encouraging due to the large size of the error projection. We do not have the source code for this program, but investigating the model reveals that there is a loop that calls into most of the code, and the error can occur inside the loop. If the loop resets its state when looping back, the error projection would include everything inside the loop or called from it. This is because for every node, there is a path from the loop head that goes through the node, then loops back to the head, with the same data state, and then goes to error.

This seems to be a limitation of error projections and perhaps calls for similar techniques that only focus on acyclic paths (paths that do not repeat a program state). However, for use inside a refinement process, error projections still give the minimal set of nodes that is sound with respect to the property being verified (focusing on acyclic paths need not be sound, i.e., the actual path that leads to error might actually be cyclic in an abstract model).

## 7 Additional Applications

The techniques presented in §4 and §5 give rise to several other applications of our ideas in model checking. Let $\mathrm{BW}(w_{\mathrm{bot}}, \gamma)$ be the weight obtained from the functional automaton intersected with $(\gamma\ \Gamma^*)$ and bottleneck weight $w_{\mathrm{bot}}$ (as used in §5). This weight can be computed for all nodes $\gamma$ in roughly the same time as the error projection (which computes $\mathrm{BW}(\overline{1}, \gamma)$).

**Multi-threaded programs.** KISS [20] is a system that can detect errors in concurrent programs that arise in at most two context switches. The two-context-switch bound enables verification using a sequential model checker. To convert a concurrent program into one suitable for a sequential model checker, KISS adds nondeterministic function calls to the `main` method of process 2 after each statement of process 1. Likewise it adds nondeterministic function returns after each statement of process 2. It also ensures that a function call from process 1 to process 2 is only performed once. This technique essentially results in a sequential program that mimics the behavior of a concurrent program for two context switches.

Using our techniques, we can extend KISS to determine all of the nodes in process 1 where a context switch can occur that leads to an error later in process 1. One way to do this is to use nondeterministic calls and returns as KISS does and then compute the error projection. However, due to the automata-theoretic techniques we employ, we can omit the extra additions. The following algorithm shows how to do this:

1. Create $\mathcal{A}_{\lhd} = \mathcal{A}_{post^*} \lhd \mathcal{A}_{pre^*}$ for process 1.
2. Let $\mathcal{A}_2$ be the result of a poststar query from `main` for process 2. Let $w = path\_summary(\mathcal{A}_2)$; $w$ represents the state transformation caused by the execution steps spent in process 2.
3. For each program node $\gamma$ of process 1, let $w_\gamma = \mathrm{BW}(w, \gamma)$ be the weight obtained from functional automaton $\mathcal{A}_{\lhd}$ of process 1. If $w_\gamma \neq \overline{0}$ then an error can occur in the program when the first context switch occurs at node $\gamma$ in process 1.

**Error reporting.** The model checker SLAM [2] used a technique presented in [1] to identify error causes from counterexample traces. The main idea was to remove "correct" transitions from the error trace and the remaining transitions indicate the cause of the error. These correct transitions were obtained by a backward analysis from non-error configurations. However, no restrictions were imposed that these transitions also be reachable from the entry point of the program. Using annotated error projections, we can limit the correct transitions to ones that are both forward reachable from program entry and backward reachable from the non-error configurations.

## 8 Related Work

The combination of forward and backward analysis has a long history in abstract interpretation, going back to Cousot's thesis [8]. It has been also been used in model checking [17] and in interprocedural analysis [13]. In the present paper, we show how forward and backward approaches can be combined precisely in the context of interprocedural analysis performed with WPDSs; our experiments show that this approach is significantly faster than a more straightforward one.

With model checkers becoming more popular, there has been considerable work on explaining the results obtained from a model checker in an attempt to localize the fault in the program [7,1]. These approaches are complimentary to ours. They build on information obtained from reachability analysis performed by the model checker and use certain heuristics to isolate the root cause of the bug. Error projections seek to maximize information that can be obtained from the reachability search so that other tools can take advantage of this gain in precision. This paper focused on using error projections inside an abstraction refinement loop. The third application in §7 briefly shows how they can be used for fault localization. It would be interesting to explore further use of error projections for fault localization.

Such error-reporting techniques have also been used outside model checking. Kremenek et al. [14] use statistical analysis to rank counterexamples found by the `xgcc`[9] compiler. Their goal is to present to the user an ordered list of counterexamples sorted by their confidence rank.

The goal of both program slicing [23] and our work on error projection is to compute a set of nodes that exhibit some property. In our work, the property of interest is membership in an error path, whereas in the case of program slicing, the property of interest is membership in a path along data and control dependences. Slicing and chopping have certain advantages—for instance, chopping filters out statements that do not transmit effects from source $s$ to target $t$. These techniques have been generalized by Hong et al. [12], who show how to perform more precise versions of slicing and chopping using predicate-abstraction and model checking. However, their methods are intraprocedural, whereas our work addresses interprocedural analysis.

Mohri et al. investigated the intersection of weighted automata in their work on natural-language recognition [18,19]. For their weight domains, the extend operation must be commutative. We do not require this restriction.

## References

1. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: POPL (2003)
2. Ball, T., Rajamani, S.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.) Model Checking Software. LNCS, vol. 2057, Springer, Heidelberg (2001)
3. Ball, T., Rajamani, S.K.: Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research (2000)
4. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, Springer, Heidelberg (1997)
5. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL (2003)
6. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. In: ICSE (2003)

7. Chaki, S., Groce, A., Strichman, O.: Explaining abstract counterexamples. In: Roy, B., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, Springer, Heidelberg (2004)
8. Cousot, P.: Méthodes itératives de construction et d'approximation de point fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes. Thèse ès sciences mathématiques, Univ. of Grenoble (1978)
9. Engler, D.R., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: OSDI (2000)
10. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. Elec. Notes in Theoretical Comp. Sci. 9 (1997)
11. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL (2002)
12. Hong, H.S., Lee, I., Sokolsky, O.: Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In: SCAM (2005)
13. Jeannet, B., Serwe, W.: Abstracting call-stacks for interprocedural verification of imperative programs. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, Springer, Heidelberg (2004)
14. Kremenek, T., Ashcraft, K., Yang, J., Engler, D.R.: Correlation exploitation in error ranking. In: SIGSOFT FSE (2004)
15. Lal, A., Kidd, N., Reps, T., Touili, T.: Abstract error projection. Technical Report 1579, University of Wisconsin-Madison (January 2007)
16. Lal, A., Reps, T.: Improving pushdown system model checking. Technical Report 1552, University of Wisconsin-Madison (January 2006)
17. Massé, D.: Combining forward and backward analyses of temporal properties. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, Springer, Heidelberg (2001)
18. Mohri, M., Pereira, F.C.N., Riley, M.: Weighted automata in text and speech processing. In: ECAI (1996)
19. Mohri, M., Pereira, F.C.N., Riley, M.: The design principles of a weighted finite-state transducer library. In: Theoretical Computer Science (2000)
20. Qadeer, S., Wu, D.: KISS: Keep it simple and sequential. In: PLDI (2004)
21. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. SCP, 58 (2005)
22. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, Tech. Univ. Munich (2002)
23. Weiser, M.: Program slicing. IEEE Trans. Software Eng. 10(4), 352–357 (1984)

# Precise Thread-Modular Verification

Alexander Malkis[1], Andreas Podelski[1], and Andrey Rybalchenko[2,3]

[1] University of Freiburg
[2] EPFL
[3] MPI

**Abstract.** Thread-modular verification is a promising approach for the verification of concurrent programs. Its high efficiency is achieved by abstracting the interaction between threads. The resulting polynomial complexity (in the number of threads) has its price: many interesting concurrent programs cannot be handled due to the imprecision of the abstraction. We propose a new abstraction algorithm for thread-modular verification that offers both high degree precision and polynomial complexity. Our algorithm is based on a new abstraction domain that combines Cartesian abstraction with *exception sets*, which allow one to handle particular thread interactions precisely. Our experimental results demonstrate the practical applicability of the algorithm.

## 1 Introduction

Many software systems are built from concurrent components. The development of such systems is a difficult and error prone task, since the programmer needs to write code that correctly handles all possible interactions between multiple concurrent threads. Verification of multi-threaded software is a hard problem [11]. The number of states of multi-threaded programs grows exponentially with the number of threads, which is called the state-explosion problem. There exist a variety of techniques and tools for the verification of multi-threaded programs, see e.g. [1, 2, 7, 6, 8, 14, 15, 16], which aim at reducing the number of states that needs to be inspected to verify a property.

One promising approach to circumvent the state explosion problem is offered by verification algorithms that reason about concurrent software modularly. Modularity allows one to avoid the explicit construction of the global state space by considering each thread in isolation, see e.g. [7, 9, 13]. The resulting polynomial complexity (in the number of threads) has its price: many interesting concurrent programs cannot be handled due to the imprecision of the abstraction [7]. For example, the existing thread-modular algorithms cannot prove the mutual exclusion property of the following simple concurrent fragment, which commonly appears in concurrent programs:

$$P_1 :: \begin{bmatrix} \ell_1 : \textbf{acquire}\ lck \\ \ell_2 : \textbf{critical} \\ \ell_3 : \textbf{release}\ lck \end{bmatrix} \quad \| \quad P_2 :: \begin{bmatrix} m_1 : \textbf{acquire}\ lck \\ m_2 : \textbf{critical} \\ m_3 : \textbf{release}\ lck \end{bmatrix}$$

Here, **acquire** *lck* waits until the lock variable *lck* becomes false and subsequently sets it to true. The call **release** *lck* sets the variable *lck* back to false.

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 218–232, 2007.

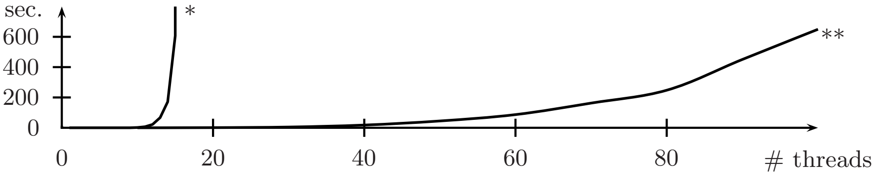


**Fig. 1.** Non-modular (∗) vs. thread-modular (∗∗) verification with exception set. We consider the example program given in Section 1 scaled w.r.t. the number of concurrent threads. Our algorithm retains polynomial complexity while gaining additional precision.

We observe that the root of the imprecision lies in the fact that the thread-modular reasoning abstracts away crucial dependencies between local states of different threads, which are necessary to establish the property.

We propose a new abstraction algorithm for thread-modular verification that offers improved precision still within polynomial complexity. Our algorithm exploits the insight that we can prevent the undesired precision loss by preserving dependencies between certain sets of local states. These dependencies would otherwise be lost due to thread-modular abstraction. Stated in terms of abstraction, we exclude some a priori fixed set of program states from the abstraction process, and always treat them concretely. We refer to such sets as *exception sets*.

We formalize the notion of exception sets and their application in thread-modular verification in the framework of abstract interpretation [3], where we define a pair of abstraction and concretization functions that implement the application of exception sets. Now, we can combine any existing abstract interpretation with our exception set-based algorithm in a modular way, following [4]. In this paper, we study the combination of exception sets and Cartesian abstraction. Our interest in this combination is naturally motivated by the fact that thread-modular verification algorithms implement Cartesian abstraction [13]. We provide efficient algorithms for abstract interpretation in the combined abstraction, which retain the polynomial run time of the reachability computation with Cartesian abstraction while gaining precision from the exception sets. We identify an interesting class of concurrent programs for which our algorithm is precise and efficient. This class is obtained by parameterizing the fragment above with respect to the number of concurrent threads and the number of critical sections in per thread.

We implemented our algorithm for precise thread-modular verification, and applied it on a series of benchmarks. The scalability of our implementation is promising: by using exception sets we were able to increase the number of concurrent threads that can be handled by our implementation by an order of magnitude, see Figure 1 and Section 6.

The main contributions of the paper consist of:

- an abstraction method with exception sets, which allows one to treat some part of the state space without abstraction;
- an implementation of the exception set-based abstraction with polynomial complexity in the number of threads and in the description of the exception set;

- an identification of a class of programs that allow verification in fully polynomial time;
- an experimental evaluation of a set of benchmarks that provides practical evidence for scalability of a prototype implementation.

The rest of the paper is organized as follows. First, we illustrate our approach for precise thread-modular verification with exception sets on a simple example. Section 3 formalizes abstraction and concretization with exception sets. In Section 4, we formally describe the verification algorithm. We present the class of programs on which our algorithm is precise and efficient in Section 5. Section 6 describes our experimental evaluation. We discuss the related work and conclude in Section 7. Some proofs are omitted due to the lack of space, and can be found in [12].

## 2 Example: Peterson's Algorithm

We illustrate our algorithm for the precise thread-modular verification on Peterson's mutual exclusion algorithm shown in Fig. 2. We wish to verify that at most one thread is in its critical section at location $D$.

$$\textbf{global } x = y = turn = 0$$

$$P_1 :: \left[\begin{array}{l} A\colon x := 1; \\ B\colon turn := 1; \\ C\colon \textbf{while}(y \textbf{ and } turn); \\ \quad\;\; \textbf{critical} \\ D\colon x := 0;\ \textbf{goto } A; \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} A\colon y := 1; \\ B\colon turn := 0; \\ C\colon \textbf{while}(x \textbf{ and not } turn); \\ \quad\;\; \textbf{critical} \\ D\colon y := 0;\ \textbf{goto } A; \end{array}\right]$$

**Fig. 2.** Peterson's mutual exclusion algorithm

First let us compute an over-approximation of the reachable states by applying a thread-modular verification algorithm, e.g. [7]. The result is represented by the following union of Cartesian products:

$$\begin{array}{rlll} & \{000\} & \times\{A\} & \times\{A\} \\ \cup & \{001\} & \times\{A\} & \times\{A\} \\ \cup & \{010\} & \times\{A\} & \times\{A,B,C,D\} \\ \cup & \{011\} & \times\{A\} & \times\{A,B,C,D\} \\ \cup & \{100\} & \times\{B,C,D\} & \times\{A\} \\ \cup & \{101\} & \times\{B,C,D\} & \times\{A\} \\ \cup & \{110\} & \times\{B,C,D\} & \times\{A,B,C,D\} \\ \cup & \{111\} & \times\{B,C,D\} & \times\{A,B,C,D\}\,, \end{array}$$

where $abc$ (e.g. 011) denotes the shared part $x = a \wedge y = b \wedge turn = c$ (e.g. $x = 0 \wedge y = 1 \wedge turn = 1$). This over-approximation is too coarse. It contains some states where both the first and the second thread are at their locations $D$, namely $(111, D, D)$ and $(110, D, D)$ (i.e. $x = y = turn = 1 \wedge pc_1 = pc_2 = D$ and $x = y = 1 \wedge turn = 0 \wedge pc_1 = pc_2 = D$).

Now we apply our algorithm instead. It iteratively computes an over-approximation of the reachable states, without losing the dependencies between the successors of the states contained in a given exception set. Let $E = \{(110, B, D), (110, C, C), (111, D, B)\}$ be the exception set.

We start the iteration with the initial state set $X_0 = \{(000, A, A)\}$. We compute the over-approximation of the set of states that are reachable from it in one step, as follows. First, we take the smallest Cartesian product that contains $(000, A, A)$. This is again

$$\{000\} \times \{A\} \times \{A\}\,.$$

Then we make a step which is specific to our algorithm. We extend this set by adding the elements of the exception set, which yields

$$X_1 = \{(000, A, A), (110, B, D), (110, C, C), (111, D, B)\}\,.$$

For this set, we compute the image under the one-step reachability under post, which is induced by the program, and add the initial element, which is in $X_0$. The resulting set is $\{(000, A, A), (010, A, B), (011, A, B), (100, B, A), (110, C, C), (110, D, C), (111, C, D)\}$. Before over-approximating this set, we perform another step that is specific to our algorithm. We subtract the exception set from the result, which yields the set $\{(000, A, A), (010, A, B), (011, A, B), (100, B, A), (110, D, C), (111, C, D)\}$. Then, for each shared part, we take the smallest Cartesian product that contains the local parts. This gives again the same set

$$\begin{aligned} &\{000\} \times\{A\} \times\{A\} \\ \cup\, &\{010\} \times\{A\} \times\{B\} \\ \cup\, &\{011\} \times\{A\} \times\{B\} \\ \cup\, &\{100\} \times\{B\} \times\{A\} \\ \cup\, &\{110\} \times\{D\} \times\{C\} \\ \cup\, &\{111\} \times\{C\} \times\{D\}\,. \end{aligned}$$

At last, we restore the states which get excluded before over-approximations, obtaining

$$\begin{aligned} X_2 = \{&(000, A, A), (010, A, B), (011, A, B), (100, B, A), (110, B, D), \\ &(110, C, C), (110, D, C), (111, C, D), (111, D, B)\}\,. \end{aligned}$$

We continue the fixpoint computation by applying the standard steps interleaved with the specific steps. The standard steps are taking one-step-successors and adding the initial states. The specific steps are subtracting the exception set away, applying the over-approximation and adding the exception set back.

The fixpoint of the described procedure is

$$\begin{array}{llll}
X_4 = & \{000\} & \times\{A\} & \times\{A\} \\
\cup & \{001\} & \times\{A\} & \times\{A\} \\
\cup & \{010\} & \times\{A\} & \times\{B,C,D\} \\
\cup & \{011\} & \times\{A\} & \times\{B\} \\
\cup & \{100\} & \times\{B\} & \times\{A\} \\
\cup & \{101\} & \times\{B,C,D\} & \times\{A\} \\
\cup & \{110\} & \times\{B,D\} & \times\{B,C\} \\
\cup & \{111\} & \times\{B,C\} & \times\{B,C,D\} \\
\multicolumn{4}{l}{\cup \{(110,B,D),(110,C,C),(111,D,B)\}\,.}
\end{array}$$

It is an inductive invariant of the program. Note that this over-approximation doesn't contain a state of the form $(\_\_\_, D, D)$, so mutual exclusion is proven.

## 3 Abstraction with Exception

In this section, we formalize the notion of exception set in the framework of abstract interpretation [3]. In this setting, an exception set corresponds to an element $E$, called exception element, of the concrete domain $D$ such that $E$ is excluded from the abstraction. Additionally, we also exclude all concrete elements that are smaller than $E$ from the abstraction, which follows the intuition that any subset of the exception set should also be excluded from the abstraction.

Let $(D, \subseteq)$ be a complete Boolean lattice and $(D^{\#}, \sqsubseteq)$ be a complete lattice. Let $E$ be an exception element, and $E^c$ be its complement. We define "exceptional abstraction" and "exceptional concretization" maps

$$\alpha_E : D \to D\ , \quad \alpha_E(X) = X \cap E^c\ ,$$

and

$$\gamma_E : D \to D\ , \quad \gamma_E(X) = X \cup E\ .$$

**Proposition 1.** *The pair $(\alpha_E, \gamma_E)$ is a Galois Connection. Formally:*

$$\forall\, X, Y \in D : \quad \alpha_E(X) \subseteq Y \Leftrightarrow X \subseteq \gamma_E(Y)\,.$$

Let $(\alpha, \gamma)$ be a Galois Connection with $\alpha : D \to D^{\#}$ and $\gamma : D^{\#} \to D$ such that $\gamma$ maps the bottom of $D^{\#}$ to the bottom of $D$. The composition $(\alpha \circ \alpha_E, \gamma_E \circ \gamma)$ of the Galois Connections is again a Galois Connection. Let init $\in D$ be any element, and $F$ be a monotone function. We obtain an abstract interpretation algorithm that combines the abstraction $(\alpha, \gamma)$ with exception sets by computing the least fixpoint

$$\mathit{lfp}\ (\lambda Y.\, \alpha \circ \alpha_E(\mathrm{init} \cup F \circ \gamma_E \circ \gamma(Y))).$$

The concretization of this least fixpoint computed by applying $\gamma_E \circ \gamma$ over-approximates the least fixpoint of $\lambda x.\, \mathrm{init} \cup Fx$. Choosing $E$ as its postfixed point (i.e. init $\subseteq E$ and $FE \subseteq E$) makes this concretization equal to this postfixed fixpoint. So it is even possible to get exactly the least fixpoint of $\lambda x.\, \mathrm{init} \cup Fx$.

# 4 Precise Thread-Modular Verification

In this section, we formally present our method for thread-modular verification of multi-threaded programs, which uses exception sets for preserving dependencies between local states of different threads. We first describe multi-threaded programs. Then we provide necessary details on Cartesian abstraction, which is a basis for thread-modular verification. Finally, we describe how Cartesian abstraction can be efficiently combined with exception sets.

## 4.1 Multi-threaded Programs

A *multi-threaded program* is a tuple

$$(\mathrm{Glob}, \mathrm{Loc}, (\rightarrow_i)_{i=1}^{n}, \mathrm{init}),$$

where Glob and Loc are any sets, each $\rightarrow_i$ is a subset of $(\mathrm{Glob} \times \mathrm{Loc})^2$ (for $1 \leq i \leq n$) and $\mathrm{init} \subseteq \mathrm{Glob} \times \mathrm{Loc}^n$.

The meaning of different components of the multi-threaded program is the following:

- Loc contains valuations of local variables (including the program counter) of any thread, we call it the *local store* of the thread (without loss of generality we assume that all threads have equal local stores);
- Glob contains valuations of shared variables, we call it the *global store*;
- the elements of $\mathrm{States} = \mathrm{Glob} \times \mathrm{Loc}^n$ are called *program states*, the elements of $Q = \mathrm{Glob} \times \mathrm{Loc}$ are called *thread states*, the projection on the global store and the $i$th local store is the map

$$\pi_{\{0,i\}} : 2^{\mathrm{States}} \rightarrow 2^{Q}, \quad S \mapsto \{(g, l_i) \mid (g, l) \in S\};$$

- the relation $\rightarrow_i$ is a transition relation of the $i$th thread ($1 \leq i \leq n$);
- init is a set of initial states.

The program is equipped with the usual interleaving semantics. This means that if a thread makes a step, then it may change its own local variables and the global variables but may not change the local variables of another thread; a step of the whole program is a step of some of the threads. The successor operation maps a set of program states to the set of their successors:

$$\begin{aligned} &\mathrm{post} : 2^{\mathrm{States}} \rightarrow 2^{\mathrm{States}} \\ S \mapsto \{(g', l') \in \mathrm{States} \mid \exists\, (g, l) \in S,\, i \in \{1, ..., n\} : (g, l_i) \rightarrow_i (g', l'_i) \\ &\qquad\qquad \text{and } \forall\, j \neq i : l_j = l'_j\}. \end{aligned}$$

We are interested in proving safety properties of multi-threaded programs. Each safety property can be encoded as a reachability property and each reachability property can be encoded as reachability between a pair of states. So we are interested in whether there is a computation of any length $k \geq 0$ that starts in an initial state and ends in a single user-given error state $f$, formally:

$$\exists\, k \geq 0 : \; f \in \mathrm{post}^k(\mathrm{init})\,.$$

The state explosion problem in context of multi-threaded programs amounts to the fact that the number of program states is exponentially large in the number of threads $n$. We don't address the problem of growing state space due to the number of variables, which is also common to sequential programs.

### 4.2 Cartesian Abstract Interpretation

Thread-modular verification applies Cartesian abstraction to achieve polynomial complexity [13]. We briefly describe the necessary definitions below.

We present a concrete and an abstract domain and a Galois Connection between them that allows us to do abstract fixpoint checking. The definitions below extend the standard notion of the dependence-free abstraction [5]:

$D = 2^{\text{States}}$ is the set underlying the concrete lattice,
$D^{\#} = (2^{\text{Glob}\times\text{Loc}})^n$ is the set underlying the abstract lattice,
$\alpha_{\text{cart}}: \quad D \to D^{\#}$,
$\alpha_{\text{cart}}(S) = \big(\pi_{\{0,i\}}S\big)_{i=1}^{n}$

is the abstraction map, which projects a set of program states to the tuple of sets of thread states, so that the $i$th component of a tuple contains all states of the $i$th thread that occur in the set of program states.

$\gamma_{\text{cart}}: \quad D^{\#} \to D$,
$\gamma_{\text{cart}}(T) = \{(g,l) \mid \forall\, i \in \{1,...,n\} : (g,l_i) \in T_i\}$

is the concretization map that combines a tuple of sets of thread states to a set of program of states by putting only those thread states together that have equal global part.

The ordering on the concrete domain $D$ is inclusion, the least upper bound is the union $\cup$, the greatest lower bound is the intersection $\cap$, the complement $X^c$ of a set $X$.

The ordering on the abstract domain $D^{\#}$ is the product ordering, i.e. $T \sqsubseteq T'$ if and only if $T_i \subseteq T'_i$ for all $i \in \{1,...,n\}$. The least upper bound $\sqcup$ is componentwise union, the greatest lower bound $\sqcap$ is componentwise intersection. Thus the abstract lattice is complete. The bottom element is the tuple of empty sets $\bot = (\emptyset)_{i=1}^{n}$.

The pair of maps $(\alpha_{\text{cart}}, \gamma_{\text{cart}})$ is a Galois Connection, i.e. all $S \in D, T \in D^{\#}$ satisfy

$$\alpha_{\text{cart}}(S) \sqsubseteq T \text{ iff } S \subseteq \gamma_{\text{cart}}(T)\,.$$

### 4.3 Exception Set as Union of Maximal Cartesian Products

Our implementation of Cartesian abstraction combined with exception sets requires a suitable data structure for the representation of elements of the concrete and abstract domains. We analyze the representation of sets of tuples by sets of Cartesian products, which leads to a polynomial implementation, see Corollary 11.

We proceed by introducing some auxiliary propositions. Let $D$ be any complete lattice with order $\leq$. Let us fix some "generating" subset of $D$ so that each element of $D$ can be written as a join of some elements of the generating subset. Further let $Y \subseteq D$ be any set that contains the generating set so that the supremum of each chain in $Y$ belongs to $Y$.

For $a \in D$, an element $y \in D$ is called *a-maximal*, if it is in $Y$, is less than or equal to $a$ and there is no other greater element of $Y$ that is less than or equal to $a$, formally:

$$y \in Y \text{ and } y \leq a \text{ and } \neg\exists\, y' \in Y : y < y' \leq a\,.$$

A set $M$ is called *maximized*, if

$$M \subseteq Y \text{ and } (\forall\, y \in Y : y \leq \bigvee M \Rightarrow \exists\, y' \in M : y \leq y')\,.$$

**Proposition 2.** *Let $a \in D$. Then any element of $Y$ less than or equal to $a$ is less than or equal to some $a$-maximal element.*

**Proposition 3.** *Each element $a$ of the lattice can be represented as a join of a unique maximized antichain. This maximized antichain contains exactly the $a$-maximal elements.*

**Proposition 4.** *Each maximized set contains the unique maximized antichain with the same join. Formally:*

$$\forall \textit{ maximized } A \subseteq Y \;\; \exists^1 M \subseteq A : M \textit{ is a maximized antichain and} \bigvee M = \bigvee A\,.$$

Now let us consider the Cartesian products. Recall that a function is a set of pairs so that for each first component there is exactly one second component. For an index set $I$, a *Cartesian product* of sets $A_i$ $(i \in I)$ is the set of maps $\prod_{i\in I} A_i := \{f : I \to \cup_{i\in I} A_i \mid \forall\, i \in I : f(i) \in A_i\}$. For a subset of indices $J \subseteq I$ the *projection* of a subset $A \subseteq \prod_{i\in I} A_i$ on the components $J$ is $\pi_J A = \{f : J \to \cup_{j\in J} A_j \mid \exists\, g \in A : f \subseteq g\}$. A projection on a single index $i \in I$ is $\pi_i A = \{a \in A_i \mid \exists\, g \in A : (i, a) \in g\}$. For a natural number $n$, the set $A^n := \prod_{i=1}^{n} A$ is the $n$th power of $A$.

**Lemma 5.** *Let $A_i, B_i$ be sets indexed by $i \in I$. Then*

$$\prod_{i\in I} A_i \subseteq \prod_{i\in I} B_i \quad \Leftrightarrow \quad ((\forall\, i \in I : A_i \subseteq B_i) \;\; or \;\; \exists\, i \in I : A_i = \emptyset)\,.$$

For the power set $D = 2^{(\mathrm{Loc}^n)}$ of all tuples of length $n$, ordered by inclusion, $Y$ the set of all Cartesian products in $D$, and the set of singletons as a generating subset, the assumption is satisfied: singletons are Cartesian products and the union of a chain of Cartesian products is a Cartesian product.

By Proposition 3 every set of tuples can be represented as a union over a set of Cartesian products, so that no two Cartesian products from this set are comparable and this set is maximized. This is a crucial property for our representation of the exception set.

For a set of tuples $A \subseteq \mathrm{Loc}^n$, $i \in \mathbb{N}_n$ and $r \in \mathrm{Loc}$ let us call

$$A_{i,r} = \pi_{\mathbb{N}_n \setminus \{i\}} \{a \in A \mid a_i = r\}$$

a *restriction* of $A$ (with parameters $i, r$). An $(n-1)$-tuple lies in this set exactly if, whenever $r$ would be inserted at the $i$th position, the tuple would lie in $A$. Since projection is monotonic, restrictions are monotonic also, i.e.

$$\forall\, A \subseteq B \subseteq \mathrm{Loc}^n, i \in \mathbb{N}_n, r \in \mathrm{Loc} : \; A_{i,r} \subseteq B_{i,r} \,.$$

**Lemma 6.** *Let a set $A \subseteq \mathrm{Loc}^n$ be represented as a maximized antichain $M$ of Cartesian products. Then for each $i \in \mathbb{N}_n, r \in \mathrm{Loc}$, the restriction $A_{i,r}$ has a representation as a union of a maximized antichain $M'$ of Cartesian products with no greater cardinality than $|M|$. If* Loc *is finite and Cartesian products are stored componentwise, the elements of the new maximized antichain can be computed in polynomial time in $n$,* $|\mathrm{Loc}|$ *and $|M|$.*

In the following, we reduce the problem of computing the abstract parameterized post to a simpler problem about the "standard" Cartesian abstraction and concretization maps:

$$\alpha_c : 2^{(\mathrm{Loc}^n)} \to (2^{\mathrm{Loc}})^n, \quad \alpha_c(S) = (\pi_i S)_{i=1}^n \,,$$
$$\gamma_c : (2^{\mathrm{Loc}})^n \to 2^{(\mathrm{Loc}^n)}, \quad \gamma_c(T) = \prod_{i=1}^n T_i \,.$$

We call the elements of $(2^{\mathrm{Loc}})^n$ *Cartesian abstract* elements. The set of Cartesian products in $\mathrm{Loc}^n$ can be injected into the set of Cartesian abstract elements: a nonempty Cartesian product is bijectively mapped to the tuple of its components, the empty Cartesian product can be mapped to any tuple of sets among which at least one set is empty (for $n > 0$).

For the rest of this section we assume that the local store is finite. Each element of $(2^{\mathrm{Loc}})^n$ is represented as a list of $n$ entries, each entry is itself a list of some elements from Loc.

**Proposition 7.** *The question whether a Cartesian product is a subset of a set of tuples can be solved in polynomial time.*

*Formally: there is an algorithm that computes the map*

$$2^{(\mathrm{Loc}^n)} \times (2^{\mathrm{Loc}})^n \to \mathrm{Bool}, \; (E, A) \mapsto \gamma_c A \overset{?}{\subseteq} E$$

*where $E$ is represented as a set $M$ of Cartesian abstract elements so that $\gamma_c M$ is a maximized antichain and $E = \bigcup \gamma_c M$, in polynomial time in $|M|$, $n$ and* $|\mathrm{Loc}|$.

**Proposition 8.** *The smallest Cartesian product that contains another Cartesian product without an exception set can be computed in polynomial time.*

*Formally: there is an algorithm that computes the map*

$$2^{(\mathrm{Loc}^n)} \times (2^{\mathrm{Loc}})^n \to (2^{\mathrm{Loc}})^n,\ (E, A) \mapsto \alpha_c(E^c \cap \gamma_c A)$$

*where $E$ is represented as a set $M$ of Cartesian abstract elements so that $\gamma_c M$ is a maximized antichain and $E = \bigcup \gamma_c M$, in polynomial time in $|M|$, $n$ and $|\mathrm{Loc}|$.*

*Proof.* Let $E \subseteq \mathrm{Loc}^n$, $A \in (2^{\mathrm{Loc}})^n$. If $\gamma_c A$ is empty (which holds iff $A_i = \emptyset$ for some $i \in \mathbb{N}_n$), then the return value is the tuple of empty sets. Otherwise all $A_i$ are nonempty.

*Claim*: All $r \in \mathrm{Loc}, i \in \mathbb{N}_n$ satisfy the equivalence:

$$r \in (\alpha_c(E^c \cap \gamma_c A))_i \quad \Leftrightarrow \quad r \in A_i \text{ and } \prod_{j \in \mathbb{N}_n \setminus \{i\}} A_j \not\subseteq E_{i,r}\,.$$

To prove the "⇒" direction, let $r \in (\alpha_c(E^c \cap \gamma_c A))_i = \pi_i(E^c \cap \gamma_c A)$. So there is an $n$-tuple $a \in E^c \cap \prod_{i=1}^n A_i$ with $a_i = r$, thus $r \in A_i$. Moreover $a \notin E$ and $a_j \in A_j$ $(j \in \mathbb{N}_n)$. So the $(n-1)$-tuple $a \setminus \{(i,r)\} \in \prod_{j \in \mathbb{N}_n \setminus \{i\}} A_j$, but $a \setminus \{(i,r)\} \notin E_{i,r}$.

To prove "⇐", let $r \in A_i$ and let $a$ be an $(n-1)$-tuple with $a \in \prod_{j \in \mathbb{N}_n \setminus \{i\}} A_j$ and $a \notin E_{i,r}$. Then $a \cup \{(i,r)\} \notin E$, but $a \cup \{(i,r)\} \in \prod_{j=1}^n A_j = \gamma_c A$. Thus $a \cup \{(i,r)\} \in E^c \cap \gamma_c A$, hence $r \in \pi_i(E^c \cap \gamma_c A) = (E^c \cap \gamma_c A)_i$.

The claim is proven. By Lemma 6, for each $i \in \mathbb{N}_n, r \in \mathrm{Loc}$, there is a maximized antichain $M'_{i,r}$ of Cartesian products with union $E_{i,r}$ and componentwise representation of Cartesian products as Cartesian abstract elements, computed in polynomial time. Since $M'_{i,r}$ is maximized, $A' \subseteq M'_{i,r}$ if and only if $\exists\, C \in M'_{i,r} :\ A' \subseteq C$ for any Cartesian product $A'$, especially for $\prod_{j \in \mathbb{N}_n \setminus \{i\}} A_j$. So all $r \in \mathrm{Loc}, i \in \mathbb{N}_n$ satisfy the equivalence:

$$r \in (\alpha_c(E^c \cap \gamma_c A))_i \qquad \Leftrightarrow \qquad r \in A_i \text{ and } \forall\, C \in M'_{i,r} : \prod_{j \in \mathbb{N}_n \setminus \{i\}} A_j \not\subseteq C\,.$$

Since $M'_{i,r}$ is generated in polynomial time and inclusion of Cartesian products is polynomial-time by Lemma 5, all the components of the abstract element $\alpha_c(E^c \cap \gamma_c A)$ are computable in polynomial time. □

Now we go over to the domains used in program analysis, namely to $D = 2^{\mathrm{States}} = 2^{\mathrm{Glob} \times \mathrm{Loc}^n}$ and $D^\# = (2^{\mathrm{Glob} \times \mathrm{Loc}})^n$.

**Proposition 9.** *The smallest abstract element that is greater than or equal to the concretization of another abstract element without an exception set can be computed in polynomial time.*

*Formally: Assume that for $E \in D$, each $\{l \mid (g,l) \in E\}$ (for $g \in \mathrm{Glob}$) is represented a set of Cartesian abstract elements whose concretizations form a maximized antichain and have $\{l \mid (g,l) \in E\}$ (for $g \in \mathrm{Glob}$) as a union. Then computing the map*

$$D \times D^\# \to D^\#, \quad (E, A) \mapsto \alpha_{\mathrm{cart}}(E^c \cap \gamma_{\mathrm{cart}} A)$$

*needs polynomial time in* $n$, $|\mathrm{Loc}|$, $|\mathrm{Glob}|$ *and the maximum cardinality of an antichain.*

*Proof.* Let $A \in D^{\#}$ and $E \in D$. For each $g \in \mathrm{Glob}$ and $i \in \mathbb{N}_n$ let $A_i^{[g]} := \{l \mid (g,l) \in A_i\}$ and $A^{[g]} := \prod_{i \in \mathbb{N}_n} A_i^{[g]}$. For all $g \in \mathrm{Glob}$, $l \in \mathrm{Loc}^n$ we have: $((g,l) \in \gamma_{\mathrm{cart}} A)$ iff $(\forall\, i \in \mathbb{N}_n : (g, l_i) \in A_i)$ iff $(\forall\, i \in \mathbb{N}_n : l_i \in A_i^{[g]})$ iff $((g,l) \in \{g\} \times \prod_{i=1}^n A_i^{[g]} = \{g\} \times A^{[g]})$. Thus

$$\gamma_{\mathrm{cart}} A = \bigcup_{g \in \mathrm{Glob}} \left( \{g\} \times A^{[g]} \right) . \tag{1}$$

For $g \in \mathrm{Glob}$, let $E^{(g)} = \{l \mid (g,l) \in E\}$. Any $g \in \mathrm{Glob}$ and $B \subseteq \mathrm{Loc}^n$ satisfy the equality:

$$(\{g\} \times B) \setminus E = \{g\} \times \left( B \setminus E^{(g)} \right) . \tag{2}$$

The map

$$\beta : \mathrm{Glob} \times \left(2^{\mathrm{Loc}}\right)^n \to D^{\#}, \quad (g, (B_i)_{i=1}^n) \mapsto (\{g\} \times B_i)_{i=1}^n$$

makes abstract elements from Cartesian abstract elements and is computable in polynomial time. Any $B \subseteq \mathrm{Loc}^n$ satisfies the equation:

$$\alpha_{\mathrm{cart}}(\{g\} \times B) = (\{(g, l_i) \mid l \in B\})_{i=1}^n = (\{g\} \times \pi_i B)_{i=1}^n = \beta\,(g, \alpha_c B) . \tag{3}$$

Now

$$\alpha_{\mathrm{cart}}\,(E^c \cap \gamma_{\mathrm{cart}} A) \overset{(1)}{=} \alpha_{\mathrm{cart}} \left( E^c \cap \bigcup_{g \in \mathrm{Glob}} \left( \{g\} \times A^{[g]} \right) \right) = [\text{distributivity}]$$

$$\alpha_{\mathrm{cart}} \left( \bigcup_{g \in \mathrm{Glob}} \left( \left( \{g\} \times A^{[g]} \right) \setminus E \right) \right) = [\text{abstraction map is a join-morphism}]$$

$$\bigsqcup_{g \in \mathrm{Glob}} \alpha_{\mathrm{cart}} \left( \left( \{g\} \times A^{[g]} \right) \setminus E \right) \overset{(2)}{=} \bigsqcup_{g \in \mathrm{Glob}} \alpha_{\mathrm{cart}} \left( \{g\} \times \left( A^{[g]} \setminus E^{(g)} \right) \right) \overset{(3)}{=}$$

$$\bigsqcup_{g \in \mathrm{Glob}} \beta \left( g, \alpha_c \left( \left( \gamma_c \left( A_i^{[g]} \right)_{i=1}^n \right) \setminus E^{(g)} \right) \right) .$$

From Prop. 8 we know that $\alpha_c \left( \left( \gamma_c \left( A_i^{[g]} \right)_{i=1}^n \right) \setminus E^{(g)} \right)$ is computable in polynomial time in $n$ and $|\mathrm{Loc}|$ and the maximum cardinality of an antichain; the map $\beta$ is also polynomial and the abstract join is also polynomial. □

**Proposition 10.** *Computing the best abstract post with exceptions takes polynomial time.*

*Formally: Assume that for $E \in D$, each $\{l \mid (g,l) \in E\}$ (for $g \in \mathrm{Glob}$) is represented as a set of Cartesian abstract elements whose concretizations form a maximized antichain and have $\{l \mid (g,l) \in E\}$ as the union. Then computing the map*

$$D \times D^{\#} \to D^{\#}, (E, A) \mapsto \mathrm{post}_{E,\mathrm{cart}} A$$

*takes polynomial time in $n$, $|\mathrm{Loc}|$, $|\mathrm{Glob}|$ and the maximum size of an antichain from the representation of $E$.*

**Corollary 11.** *Computing the least abstract fixpoint with exceptional Cartesian abstraction and representation of $E$ so that each $\{l \mid (g,l) \in \mathrm{Loc}\}$ is a union of Cartesian products needs polynomial time.*

*Formally: Assume that for $E \in D$, each $\{l \mid (g,l) \in E\}$ (for $g \in \mathrm{Glob}$) is represented as a set of Cartesian abstract elements whose concretizations form a maximized antichain and have $\{l \mid (g,l) \in E\}$ as the union. Then computing the map*

$$D \times D \to D^{\#}, \quad (E, \mathrm{init}) \mapsto \mathit{lfp}\ (\lambda X.\alpha_{\mathrm{cart}}\alpha_E(\mathrm{init} \cup \mathrm{post}\gamma_E\gamma_{\mathrm{cart}}X))$$

*needs polynomial time in $n$, $|\mathrm{Loc}|$, $|\mathrm{Glob}|$, in the cardinality of the largest antichain and in $|\mathrm{init}|$.*

Note that if initial states are represented the same way as the exception set then the run time is polynomial in the cardinality of the largest antichain from the representation of init instead of |init|.

The whole algorithm can be viewed as a reduction to a polynomial number of queries of the form “is a Cartesian product a subset of a fixed set” as in Prop. 7. Each such query can be trivially answered given the representation of the fixed set as a union of all maximal (w.r.t. inclusion) Cartesian products inside this set.

## 5 Efficiently Handled Class

In this section, we describe a class of programs that can be efficiently verified by our thread-modular verification algorithm with exception sets.

Each program in the class is generated by instantiating the schema shown in Figure 3 with a fixed number $n$ of threads and a fixed number $m$ of critical sections per thread.

Let the sets of locations *InCrit* contain all local states at critical locations and *NotInCrit* be its complement:

$$\begin{aligned} \mathit{InCrit} &= \{l \in \mathrm{Loc} \mid \exists k : l(pc) \in \{\ell_2^k, \ell_3^k\}\} \\ \mathit{NotInCrit} &= \mathrm{Loc} \setminus \mathit{InCrit} \end{aligned}$$

Further for $1 \le i \le n$ let

$$C(i) = \mathit{NotInCrit}^{i-1} \times \mathit{InCrit} \times \mathit{NotInCrit}^{n-i},$$

$$P_1 :: \begin{bmatrix} \ell_1^1 : \textbf{acquire } lck \\ \ell_2^1 : \textbf{critical} \\ \ell_3^1 : \textbf{release } lck \\ \vdots \\ \ell_1^m : \textbf{acquire } lck \\ \ell_2^m : \textbf{critical} \\ \ell_3^m : \textbf{release } lck \end{bmatrix} \quad \| \quad \cdots \quad \| \quad P_n :: \begin{bmatrix} \ell_1^1 : \textbf{acquire } lck \\ \ell_2^1 : \textbf{critical} \\ \ell_3^1 : \textbf{release } lck \\ \vdots \\ \ell_1^m : \textbf{acquire } lck \\ \ell_2^m : \textbf{critical} \\ \ell_3^m : \textbf{release } lck \end{bmatrix}$$

**Fig. 3.** Schema for programs consisting of $n$ concurrent threads with $m$ critical sections per thread, which admit efficient and precise thread-modular verification

and

$$M = \{C(i) \mid i \in \mathbb{N}_n\} .$$

One can show that $M$ is a maximized antichain. Now we choose

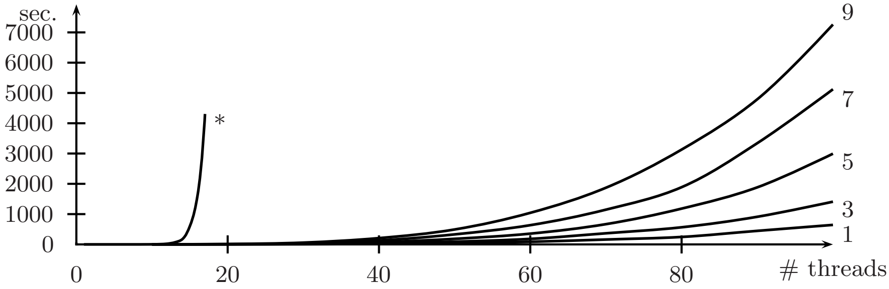
$$E = \bigcup_{g \in \text{Glob}, g(lck) \neq 0, C \in M} \{g\} \times C$$

as an exception set. Checking the abstract fixpoint computed by parameterized thread-modular algorithm proves mutual exclusion. Moreover, all antichains in the representation of $E$ have linear cardinality in $n$, so our algorithm consumes polynomial time and space. We conclude that no state explosion occurs during the application of our thread-modular algorithm.

## 6 Experiments

In this section we describe our experimental evaluation. We implemented the algorithm described in Section 4 in OCaml by using the ordered set data structure from the standard library to represent sets. We applied our implementation on a set of benchmark programs that we obtained by instantiating the schema shown in Figure 3. We experimented with the number of threads ranging from 10 to 100. For each thread size, we run our tool on programs with 1, 3, 5, 7, and 9 critical sections per thread. The resulting run times, which we obtained on 2.8 Ghz CPU, are shown in Figure 4.

We observe that our theoretical claims are supported by the experiments. The run time grows polynomially in the number of threads and in the number of critical sections. This allows us to verify instances of the program that are far beyond the reach of the algorithm that performs reachability computation without abstraction. Note that no existing thread-modular algorithm can handle the benchmark programs, due to the lack of precision.



| | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.1 | 1.2 | 5.6 | 19 | 45 | 88 | 163 | 249 | 452 | 650 |
| 3 | 0.2 | 2.8 | 13 | 40 | 90 | 184 | 370 | 570 | 920 | 1423 |
| 5 | 0.3 | 5 | 26 | 72 | 186 | 359 | 675 | 1192 | 1901 | 3022 |
| 7 | 0.6 | 8.7 | 36 | 131 | 335 | 642 | 1158 | 1907 | 3374 | 5170 |
| 9 | 0.9 | 13.9 | 60 | 210 | 498 | 1052 | 1889 | 3160 | 4836 | 7328 |

**Fig. 4.** Experimental evaluation for the number of identical concurrent threads ranging between 10 and 100, and number of critical sections per thread from the set $\{1, 3, 5, 7, 9\}$. The table contains the run times, in seconds, for different combinations of number of critical sections/threads. The curve $*$ shows the run time for the exhaustive state exploration without abstraction for a single critical section (per thread), and puts the scale into perspective.

## 7 Related Work and Conclusion

Cartesian abstraction, which is also known as "independent attribute method", is a classical abstraction means in program analysis [10]. To the best of our knowledge, our application of Cartesian abstraction for the analysis of multithreaded programs has not been known before.

The thread-modular verification algorithm of [7] serves as a starting point of our research, with the goals of improving its precision while retaining the polynomial complexity. The relationship between the thread-modular algorithm [7] and Cartesian abstraction provided a basis for the integration of exception sets into the abstraction framework.

In this paper, we presented a thread-modular verification algorithm that offers the polynomial complexity of the existing thread-modular approaches and increased precision. Such combination allows one to verify new classes of concurrent programs. Our experimental evaluation of the algorithm has shown its promising applicability.

The proposed algorithm is parameterized by an exception set, which determines the set of states that are excluded from the abstraction. We are currently developing an algorithm that computes an adequate exception set automatically. One possible direction is shown in Section 5.

## Acknowledgements

The first author is supported in part by a DFG-Graduiertenkolleg of the University of Freiburg. The third author is supported in part by Microsoft Research through the European Fellowship Programme. We would like to thank anonymous reviewers for comments and suggestions.

## References

1. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. Int. J. Found. Comput. Sci. 14(4), 551–582 (2003)
2. Clarke, E.M., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 126–141. Springer, Heidelberg (2005)
3. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
4. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282 (1979)
5. Cousot, P., Cousot, R.: Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In: FPCA, pp. 170–181 (1995)
6. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL'2005, pp. 110–121. ACM Press, New York (2005)
7. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: Ball, T., Rajamani, S.K. (eds.) Model Checking Software. LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003)
8. Henzinger, T.A., Jhala, R., Majumdar, R.: Race checking by context inference. In: PLDI'2004, pp. 1–13. ACM Press, New York (2004)
9. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread-modular abstraction refinement. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 262–274. Springer, Heidelberg (2003)
10. Jones, N.D., Muchnick, S.S.: Complexity of flow analysis, inductive assertion synthesis and a language due to dijkstra. In: FOCS, pp. 185–190. IEEE (1980)
11. Kozen, D.: Lower bounds for natural proof systems. In: FOCS, pp. 254–266. IEEE Computer Society Press, Rhode Island (1977)
12. Malkis, A., Podelski, A., Rybalchenko, A.: Precise thread-modular verification with exception sets, technical report (2006), http://www.mpi-inf.mpg.de/∼malkis/report-cav07.ps
13. Malkis, A., Podelski, A., Rybalchenko, A.: Thread-modular verification is cartesian abstract interpretation. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 183–197. Springer, Heidelberg (2006)
14. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
15. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. In: PLDI'2004, pp. 14–24. ACM Press, New York (2004)
16. Robby, Dwyer, M.B., Hatcliff, J.: Bogor: an extensible and highly-modular software model checking framework. In: ESEC / SIGSOFT FSE, pp. 267–276 (2003)

# Modular Safety Checking for Fine-Grained Concurrency

Cristiano Calcagno[1], Matthew Parkinson[2], and Viktor Vafeiadis[2]

[1] Imperial College, London
[2] University of Cambridge

**Abstract.** Concurrent programs are difficult to verify because the proof must consider the interactions between the threads. Fine-grained concurrency and heap allocated data structures exacerbate this problem, because threads interfere more often and in richer ways. In this paper we provide a thread-modular safety checker for a class of pointer-manipulating fine-grained concurrent algorithms. Our checker uses ownership to avoid interference whenever possible, and rely/guarantee (assume/guarantee) to deal with interference when it genuinely exists.

## 1 Introduction

Traditional concurrent implementations use a single synchronisation mechanism, such as a lock, to guard an entire data structure (such as a list or a hash table). This *coarse-grained* synchronisation makes it relatively easy to reason about correctness, but it limits concurrency, negating some of the advantages of modern multi-core or multi-processor architectures. A *fine-grained* implementation permits more concurrency by allowing multiple threads to access the same data structure simultaneously. Of course, this makes it far harder to reason about correctness.

There has been a lot of reseach [9,11,1,24,29] on verifying coarse-grained concurrent programs, but hardly any on verifying fine-grained concurrency. Recently, we have presented a new logic, RGSep [28], and demonstrated its use in precisely and concisely describing the inter-thread interference of fine-grained concurrent algorithms—thus making the proof of fine-grained concurrent programs easier. The logic merges aspects from both rely/guarantee reasoning [15] and concurrent separation logic [18,5], giving rise to simple, modular proofs about algorithms with intricate concurrency and dynamic memory management.

The difficulty with RGSep [28] is that it is simply a logic: users of it must manually prove their programs correct with pen and paper. In this paper, we automated a suitable subset of RGSep and implemented a new modular tool that automatically verifies safety properties of a class of intricate concurrent algorithms.

Like ESC/Java [10] and Spec# [2], our tool symbolically simulates the code and produces verification conditions that, if proved valid, imply that the program is correct with respect to the user-supplied pre-/post-condition pair. In doing

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 233–248, 2007.

so, our tool splits the state (heap) into thread-local state and shared state. We maintain this partition throughout symbolic execution using assertions that describe the partition between the local and the shared state. The assertions are restricted to a subset of separation logic chosen to support a form of symbolic execution, a decidable proof theory for symbolic heaps, and the inference of frames for handling procedure calls [3]. Our prover starts with the precondition, symbolically executes the code deriving a postcondition, and checks that this implies the user-supplied postcondition.

In order to handle fine-grained concurrency modularly, we require the programmer to describe the interference between threads using lightweight annotations. These take the form of actions done by the program; they are much more concise than invariants and much easier to come up with. So far, our tool checks only safety properties: in particular, data integrity, memory leaks, and race-conditions.[1] It does not check liveness properties like termination.

Our technical contributions are:

- to enrich the set of separation logic operators handled automatically (see §2.1);
- a procedure for calculating the interference imposed by the environment, which, given an assertion, computes a weaker assertion that is *stable* under interference and, hence, valid to use in a rely-guarantee proof (see §2.3);
- a symbolic execution for RGSep assertions (see §2.4);
- an automatic safety checker specialised to list-manipulating programs; and
- verification of a series of fine-grained concurrent algorithms.

In Section 3, we describe the tool by example. In particular, we demonstrate a verification of a lock-coupling list algorithm, which highlights (1) dynamic lock allocation; (2) memory deallocation, including locks; and (3) non-nested locking. In Section 4, we evaluate the performance of our tool.

# 2 The Analysis

## 2.1 RGSep Assertions

Our tool splits the state (heap) into thread-local state and shared state, hence our assertions specify a state consisting of two heaps with disjoint domains: the local heap (visible to a single thread), and the shared heap (visible to all threads). Normal formulae, $P$, specify the local heap, whereas boxed formulae, $\boxed{P}$, specify the shared heap.[2] Note that boxes cannot be nested.

Our tool accepts assertions written in the following grammar:

$$\begin{aligned} A, B &::= E_1{=}E_2 \mid E_1{\neq}E_2 \mid E \mapsto \rho \mid \mathsf{lseg}(E_1, E_2) \mid \mathsf{junk} \\ P, Q, R, S &::= A \mid P \vee Q \mid P * Q \mid P \mathbin{-\!\!\circledast} Q \mid P{\downharpoonright}_{E_1,\ldots,E_n} \\ p, q &::= p \vee q \mid P * \boxed{Q} \end{aligned}$$

[1] We allow racy interference where the user specifies it, but not elsewhere.

[2] More generally, we support multiple disjoint regions of shared state, and boxes are annotated with the name $r$ of the region: $\boxed{P}_r$. For clarity of exposition, we present the analysis with respect to a single resource, and omit the subscript.

where $E$ is a pure expression, an expression that does not depend on the heap. All variables starting with an underscore (e.g., $\_\mathtt{x}$) are implicitly existentially quantified. In the assertion $P * \boxed{Q}$, if $X$ contains the set of existential free variables of $P$ and $Q$, then their scope is $\exists X.\ P * \boxed{Q}$.

The first line contains the usual atomic assertions of separation logic: pure predicates (that do not depend on the heap), heap cells ($E \mapsto \rho$), list segments ($\mathsf{lseg}(E_1, E_2)$), and $\mathsf{junk}$. $E \mapsto \rho$ asserts that the heap consists of a single memory cell with address $E$ and contents $\rho$, where $\rho$ is a mapping from field names to values (pure expressions); $\mathsf{lseg}(E_1, E_2)$ says that the heap consists of an acyclic linked list segment starting at $E_1$ and ending at $E_2$; $\mathsf{junk}$ asserts the heap may contain inaccessible state.

The second line contains operators for building larger formulae. Disjunction, $P \vee Q$, asserts that the heap satisfies $P$ or $Q$. Separating conjunction, $P * Q$, asserts that the heap can be divided into two (disjoint) parts, one satisfying $P$ and the other satisfying $Q$. For notational convenience, we let pure formulae (e.g., $E_1 = E_2$) hold only on the empty heap, and use only one connective ($*$) to express both ordinary conjunction for pure formulas[3], and the separating conjunction between heap formulas. The other two operators are new.

- *Septraction* ($-\circledast$) is defined as $h \models (P -\circledast Q) \iff \exists h_1\, h_2.\ h_2 = h * h_1$ and $h_1 \models P$ and $h_2 \models Q$. This operation can be thought of as subtraction or differentiation, as it achieves the effect of subtracting heap $h_1$ satisfying $P$ from the bigger heap $h_2$ satisfying $Q$.
- The "dangling" operator $P\lfloor_D$ asserts that $P$ holds and that all locations in the set $D$ are not allocated. This can be defined in separation logic as $P\lfloor_{(E_1,\dots,E_n)} \iff P \wedge \neg((E_1 \mapsto \_) * \mathit{true}) \wedge \cdots \wedge \neg((E_n \mapsto \_) * \mathit{true})$, but it is better treated as a built-in assertion form, because it is much easier to analyse than $\wedge$ and $\neg$.

(For formal definitions and further detail, please see the technical report on the logic [28].)

Finally, the third line introduces $P * \boxed{Q}$, the novel assertion of RGSep, which does not exist in separation logic. It asserts that the shared state satisfies $Q$ and that the local state is separate and satisfies $P$.

Extending any separation logic theorem prover to handle the dangling operator, $\lfloor_D$, and septraction, $-\circledast$, is relatively simple. The 'dangling' operator can be eliminated from all terms (see Fig. 1), except for terms containing recursive predicates, such as $\mathsf{lseg}$. Recursive predicates require the dangling set $D$ to be passed as a parameter,

$$\mathsf{lsegi}_{tl,\rho}(E_1, E_2, D) \stackrel{\text{def}}{=} (E_1 = E_2) \vee \exists x.\, E_1 {\mapsto} (tl{=}x, \rho)\lfloor_D * \mathsf{lsegi}_{tl,\rho}(x, E_2, D)$$

[3] Technically, we write for example $E_1{=}E_2$ as an abbreviation for the separation logic formula $(E_1 {=}_{\mathrm{SL}} E_2) \wedge \mathit{emp}$, so that $(E_1{=}E_2) * P$ is equivalent to $(E_1 {=}_{\mathrm{SL}} E_2) \wedge P$.

$$
\begin{aligned}
(F \mapsto \rho){\downharpoonright_D} &\iff F \neq D * (F \mapsto \rho) \\
&\quad\text{where } [F{\neq}\{E_1,\dots,E_n\} \stackrel{\text{def}}{=} F{\neq}E_1 * \cdots * F{\neq}E_n] \\
\mathsf{lsegi}_{tl,\rho}(E,F,D'){\downharpoonright_D} &\iff \mathsf{lsegi}_{tl,\rho}(E,F,D \cup D') \\
(P * Q){\downharpoonright_D} &\iff P{\downharpoonright_D} * Q{\downharpoonright_D} \\
(P \vee Q){\downharpoonright_D} &\iff P{\downharpoonright_D} \vee Q{\downharpoonright_D} \\
\\
(E_1 \mapsto \rho_1) \mathbin{-\!\!\circledast} (E_2 \mapsto \rho_2) &\iff E_1 = E_2 * \rho_1 = \rho_2 \\
(E_1 \mapsto \mathtt{tl}{=}E_2, \rho) \mathbin{-\!\!\circledast} \mathsf{lsegi}_{tl,\rho'}(E,E',D) &\iff \\
E_1{\neq}0 * E_1{\neq}D * \rho{=}\rho' * \mathsf{lsegi}_{tl,\rho'}&(E,E_1,D){\downharpoonright_{E'}} * \mathsf{lsegi}_{tl,\rho'}(E_2,E',D){\downharpoonright_{E_1}} \\
&\quad\text{where } [\rho{=}\rho' \stackrel{\text{def}}{=} \forall f \in (\mathit{dom}(\rho) \cap \mathit{dom}(\rho')).\rho(f) = \rho'(f)] \\
(E \mapsto \rho) \mathbin{-\!\!\circledast} (P * Q) &\iff P{\downharpoonright_E} * (E \mapsto \rho \mathbin{-\!\!\circledast} Q) \\
&\qquad\qquad \vee\ (E \mapsto \rho \mathbin{-\!\!\circledast} P) * Q{\downharpoonright_E} \\
(E \mapsto \rho) \mathbin{-\!\!\circledast} (P \vee Q) &\iff (E \mapsto \rho \mathbin{-\!\!\circledast} P)\ \vee\ (E \mapsto \rho \mathbin{-\!\!\circledast} Q) \\
(P * Q) \mathbin{-\!\!\circledast} R &\iff P \mathbin{-\!\!\circledast} (Q \mathbin{-\!\!\circledast} R) \\
(P \vee Q) \mathbin{-\!\!\circledast} R &\iff (P \mathbin{-\!\!\circledast} R) \vee (Q \mathbin{-\!\!\circledast} R)
\end{aligned}
$$

**Fig. 1.** Elimination rules for $P{\downharpoonright_D}$ and septraction ($\mathbin{-\!\!\circledast}$)

We subscript the list segments with the linking field, *tl*, and any common fields, $\rho$, that all the nodes in the list segment have.[4] We omit the subscript when the linking field is *tl* and $\rho$ is empty. Unlike the definition of lseg, our new definition is imprecise: $\mathsf{lsegi}(E, E, \{\})$ describes both the empty heap and a cyclic list. The imprecise definition allows us to simplify the theorem prover, as the side-conditions for appending list segments are not needed. A precise list segment, $\mathsf{lseg}(E_1, E_2)$, is just a special case our imprecise list segment, $\mathsf{lsegi}(E_1, E_2, \{E_2\})$. Another benefit of the dangling operator is that some proof rules can be strengthened, removing some causes of incompleteness. For instance, the following application of the proof rule for deallocating a memory cell $\{P * \mathtt{x}{\mapsto}\_\}$ `dispose(x)` $\{P\}$ can be strengthened by rewriting the precondition and obtain $\{P{\downharpoonright_{\mathtt{x}}} * \mathtt{x}{\mapsto}\_\}$ `dispose(x)` $\{P{\downharpoonright_{\mathtt{x}}}\}$. Similarly, we can eliminate the septraction operator from $P \mathbin{-\!\!\circledast} Q$, provided $P$ does not contain any lseg or junk predicates (see Fig. 1). Had we allowed further inductive predicates, such as $\mathsf{tree}(E_1)$, we would have needed an additional rule for computing $(E \mapsto \rho) \mathbin{-\!\!\circledast} \mathsf{tree}(E_1)$.

Finally, assertions containing boxes are always written in a canonical form, $\bigvee_i (P_i * \boxed{Q_i})$. Given an implication between formulas in this form, we can essentially check implications between normal separation logic formulae, by the following lemma:

$$(P \vdash P') \wedge (Q \vdash Q') \Longrightarrow (P * \boxed{Q} \vdash P' * \boxed{Q'})$$

Furthermore, we can deduce from $P * \boxed{Q}$ all the heap-independent facts, such as $x \neq y$, which are consequences of $P * Q$, since shared and local states are always disjoint.

[4] This is important for the `lazy list` algorithm, as the invariant involves a list where all the nodes are marked as deleted (have a `marked` field set to 1).

## 2.2 Interference Actions and Stability

RGSep assertions distinguish between the local and the shared state. Local state belongs to a single thread and cannot be accessed by other concurrent threads. Shared state, however, can be accessed by any thread; hence, the logic models interference from the other threads.

In the style of rely/guarantee, we specify interference as a binary relation between (shared) states, but represent it compactly as a set of actions (updates) to the shared state. In this paper, we do not attempt to infer such actions; instead, we provide convenient syntax for the user to define them.

Consider the following two action declarations:

```
action Lock(x)      [x|->lk=0  ]  [x|->lk=TID]
action Unlock(x)    [x|->lk=TID]  [x|->lk=0  ]
```

Each action has a name, some parameters, a precondition and a postcondition. For instance, Lock(x) takes a location x whose lk field is zero, and replaces it with TID, which stands for the current thread identifier (which is unique for each thread and always non-zero). Crucially, the precondition and the postcondition delimit the overall footprint of the action on the shared state. They assert that the action does not modify any *shared* state other than x.

We abstract the behaviour of the environment as a set of actions. We say that an assertion $S$ is stable (i.e., unaffected by interference), if and only if, for all possible environment actions Act, if $S$ holds initially and Act executes then $S$ still holds at the end. Assertions about the local state are stable by construction, because interference does not affect the local state. An assertion $\boxed{S}$ about the shared state is stable under the action $P \rightsquigarrow Q$, if and only if, the following implication holds

$$(P \mathrel{-\!\circledast} S) * Q \implies S.$$

The formula $(P \mathrel{-\!\circledast} S) * Q$ represents the result of executing the environment action $P \rightsquigarrow Q$ on the initial state $S$: that is to remove $P$ from $S$ and put back $Q$ in its place. This form of environment execution is reminiscent of the idea of execution of specification statements [17]. The crucial difference here is that we do not require the implication $S \Rightarrow (P * \mathit{true})$ to hold, which would amount to checking that the environment can execute $P \rightsquigarrow Q$ on *all* the states described by $S$.

## 2.3 Inferring Stable Assertions

Most often, the postcondition of a critical section obtained by symbolic execution is not stable under interference; therefore, we must find a stable postcondition which is weaker than the original.

Assume for the time being that the rely contains a single action Act with precondition $P$ and postcondition $Q$; later, we will address the general case.

Mathematically, inferring a stable assertion from an unstable assertion $S$ is a straightforward fixpoint computation

$$S_0 = S \qquad\qquad S_{n+1} = S_n \vee (P \mathrel{-\!\!\circledast} S_n) * Q,$$

where $S_n$ is the result of at most $n$ executions of `Act` starting from $S$. This computation, however, does not always terminate, because the assertions can contain an unbounded number of existentially quantified variables, and hence the domain is infinite.

We approximate the fixpoint by using abstract interpretation. Our concrete domain is the set of syntactic Smallfoot assertions, and our abstract domain is a finite subset of normalised Smallfoot assertions that contain a bounded number of existential variables. Both domains are lattices ordered by implication, with *true* as $\top$ and *false* as $\bot$; $\vee$ is join.

We have a lossy abstraction function $\alpha : \mathsf{Assertion} \to \mathsf{RestrictedAssertion}$ that converts a Smallfoot assertion to a restricted assertion, and a concretisation function $\gamma : \mathsf{RestrictedAssertion} \to \mathsf{Assertion}$ which is just the straightforward inclusion (i.e., the identity) function. In our implementation, the abstraction function $\alpha$ is computed by applying a set of abstraction rules, an adaptation of the rules of Distefano et al. [8]. Our abstraction function ensures that any existential variable in the final assertion is reachable from at least two normal (program) variables. Hence, as there is a finite number of program variables and field names, the number of existential variable is also finite. To do this, we convert assertions such as $x \mapsto \_y * \_y \mapsto z$ into a list segment. The details are at the end of this section. Nevertheless, the technique is parametric to any suitable abstraction function.

The fixpoint can be computed in the abstract domain as follows:

$$S_0 = \alpha(S) \qquad S_{n+1} = S_n \vee \alpha((P \mathrel{-\!\!\circledast} S_n) * Q).$$

In the general case we have $n$ actions $\mathtt{act}_1, \ldots, \mathtt{act}_n$. Two natural algorithms are to interleave the actions during the fixpoint computation, or to stabilise one action at a time. We found that the latter strategy gives better execution times.

We now present an example of stabilisation. Consider stabilising an assertion $x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}$ with the `Lock` and `Unlock` actions from Section 2.2. Before stabilising, we replace variable `TID` in the specification of the actions with a fresh existentially quantified variable $\_tid$, and add assumptions $\_tid \neq 0$ and $\_tid \neq \mathtt{TID}$. The idea is that any thread might be executing in parallel with our thread, and all we know is that the thread identifier cannot be 0 (by a design choice) and it cannot be `TID` (because `TID` is the thread identifier of our thread). Stabilisation will involve the first and third rules in Figure 1. In most cases, an inconsistent assertion would be generated by adding one of the following equalities: $0 = 1$, $0 = \_tid$, $\mathtt{TID} = \_tid$. In the following fixpoint computation we do not list those cases.

$$\begin{aligned}
S_0 &\iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \texttt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \texttt{TID}\\
&\qquad\qquad\text{action } \texttt{lock}\\
S_1 &\iff S_0 \lor \alpha(\_tid \neq 0 * \_tid \neq \texttt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \texttt{TID})\\
&\iff S_0 \lor (\_tid \neq 0 * \_tid \neq \texttt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \texttt{TID})\\
&\iff \_tid \neq \texttt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \texttt{TID}\\
&\qquad\qquad\text{action } \texttt{lock}\\
S_2 &\iff S_1 \lor \alpha(\_tid' \neq 0 * \_tid' \neq \texttt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \texttt{TID})\\
&\iff S_1 \lor (\_tid' \neq 0 * \_tid' \neq \texttt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \texttt{TID})\\
&\iff (\_tid \neq \texttt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \texttt{TID}) \iff S_1\\
&\qquad\qquad\text{action } \texttt{unlock}\\
S_3 &\iff S_2 \lor \alpha(x \mapsto lk = 0 * y \mapsto lk = \texttt{TID})\\
&\iff S_2 \lor (x \mapsto lk = 0 * y \mapsto lk = \texttt{TID})\\
&\iff S_2
\end{aligned}$$

In this case, we do not need to stabilise with respect to `lock` again, since `unlock` produced no changes.

**Adaptation of Distefano et al.'s abstraction.** We assume that variable names are ordered, so that existential variables are smaller than normal variables. We present an algorithm that abstracts a formula $P = (A_1 * \ldots A_n)$ where each $A_i$ is an atomic formula.

1. Rewrite all equalities $E = F$, so that $E$ is a single variable, which is 'smaller' than all the variables in $F$.
2. For each equality $E = F$ in $P$, substitute any other occurences of $E$ in $P$ by $F$; if $E$ is an existential variable, discard the equality.
3. For each $A_i$ describing a memory structure (i.e., a cell or a list segment) starting at an existential address, find all other terms that can point to that address. If there are none, $A_i$ is unreachable; replace it with junk. If there is only one, then try to combine them into a list segment. If there are more than one, so $A_i$ is shared, leave them as they are.

   To combine two terms into a list segment, we use the following implication:

   $$L_{\texttt{tl},\rho_1}(E_1, \_\texttt{x})\lfloor_{D_1} * L_{\texttt{tl},\rho_2}(\_\texttt{x}, E_2)\lfloor_{D_2} \implies \mathsf{lseg}_{\texttt{tl},\rho_1\cap\rho_2}(E_1, E_2)\lfloor_{D_1\cap D_2}$$

   where $L_{\texttt{tl},\rho}(E, F)\lfloor_D$ is $lseg_{\texttt{tl},\rho}(E, F)\lfloor_D$ or $E \mapsto (\texttt{tl} = F, \rho) * E \neq D$. This is a generalisation of Distefano et al.'s rule, because our list segments record common fields $\rho$ of nodes, and the set $D$ of disjoint memory locations. In addition, because our list segments are imprecise, we do not need Distefano et al's side condition that $E_2$ is `nil` or allocated separately.
4. Put the formulae in a canonical order, by renaming their existential variables. This is achieved by, first, ordering atomic formulas by only looking at their shape, while ignoring the ordering between existential variables, and then, renaming the existential variables based on the order they appear in the ordered formula.

If we simply ran this analysis as described above, we would lose too much information and could not prove even the simplest programs. This is because the analysis would abstract $x{\mapsto}(lk{=}\texttt{TID}, tl{=}\_y){*}lseg(\_y,z)$ into $lseg(x,z)$, forgetting that the node $x$ was locked! Instead, before we start the fixed point calculation, we replace existential variables in such $\mapsto$ assertions containing occurrences of TID with normal variables to stop the abstraction rules from firing. As the number of normal variables does not increase during the fixpoint computation, the analysis still terminates. At the end of the fixed point calculation, we replace them back with existential variables. Our experiments indicate that this simple heuristic gives enough precision in practice. We have also found that turning dead program variables into existential variables before starting the fixed point calculation significantly reduces the number of cases and speeds up the analysis.

### 2.4 Symbolic Execution of Atomic Blocks

We discharge verification conditions by performing a form of symbolic execution [16,3] on symbolic states, and then check that the result implies the given postcondition. Symbolic heaps are formulae of the form

$$A_1 * \ldots * A_m * \boxed{\bigvee_i B_{1,i} * \ldots * B_{n_i,i}}$$

where each $A_i$ and $B_{i,j}$ is an atomic formula. The $A$ part of a symbolic heap describes the local state of the thread, and the $B$ part – inside the box – describes the shared part. We use disjunction for boxed assertions to represent more compactly the result of stabilisation, and avoid the duplication of the shared part for each disjunct. Symbolic states are finite sets of symbolic heaps, representing their disjunction. This kind of setup is typical of work on shape analysis using separation logic [8].

We allow operations to contain non-pure (accessing the heap) expressions in guards and assignments, by translating them into a series of reads to temporary variables followed by an assignment or a conditional using pure expressions, e.g. `assume(x->tl==0)` would be translated to `temp = x->tl; assume(temp==0)`, for a fresh variable `temp`. We omit the obvious details of this translation.

Except for atomic blocks, the symbolic execution is pretty standard: the shared component is just passed around. For atomic blocks more work is needed. We describe in detail the execution of an atomic block `atomic(B) {C} as Act(x)` starting from symbolic precondition $X * \boxed{S}$. Intuitively, the command is executed atomically when the condition B is satisfied. The annotation `as Act(x)` specifies that command $C$ performs shared action `Act` with the parameter instantiated with $x$. Suppose that `Act` was declared as `action Act(x) [P] [Q]`. Our task is to find the postcondition $\Psi$ in the following Hoare triple:

$$\{X * \boxed{S}\}\ \texttt{atomic(B) C as Act(x);}\ \{\Psi\}$$

Our algorithm consists of 4 parts, corresponding to the premises of the following inference rule.

$$\frac{\{X{*}S\}\ \texttt{assume(B)}\ \{X{*}P{*}F\} \quad \{X{*}P\}\ \texttt{C}\ \{X'\} \quad X' \vdash Q{*}Y \quad \mathit{stab}(Q{*}F) = R}{\{X * \boxed{S}\}\ \texttt{atomic(B) C as Act(x);}\ \{Y * \boxed{R}\}}$$

**Step 1.** Add shared state $S$ to the local state, and call the symbolic execution and theorem prover to infer the frame $F$ such that $[X * S]\texttt{assume(B)}[X * P * F]$. This step has the dual function of checking that the action's precondition $P$ is implied, and also inferring the leftover state $F$, which should not be accessed during the execution of `C`. The symbolic execution of `assume(B)` removes cases where `B` evaluates to false. Notice that the evaluation of `B` can access the shared state. If this step fails, the action's precondition cannot be met, and we report an error.

**Step 2.** Execute the body of the atomic block symbolically starting with $X * P$. Notice that $F$ is not mentioned in the precondition: because of the semantics of Hoare triples in separation logic, this ensures that command `C` does not access the state described by $F$, as required by the specification of `Act`.

**Step 3.** Call the theorem prover to infer the frame $Y$ such that $X' \vdash Q * Y$. As before, this has the effect of checking that the postcondition $Q$ is true at the end of the execution, and inferring the leftover state $Y$. This $Y$ becomes the local part of the postcondition. If the implication fails, the postcondition of the annotated action cannot be met, and we report an error.

**Step 4.** Combine the shared leftover $F$ computed in the first step with the shared postcondition $Q$, and stabilise the result $Q * F$ with respect to the execution of actions by the environment as described in Section 2.2.

*Precision.* Similar to resource invariants of concurrent separation logic, RGSep requires that $R$ in the rule above is *precise*[19], or in the presence of memory leaks *supported.* We postulate that checking for precision is unnecessary, if we drop the rule of conjunction from RGSep, which we happen not to use in the analysis. We are investigating a formal proof of soundness for that form of RGSep.

*Read-only atomics.* We have a simplified rule for read-only atomic regions that does not require an action specification:

$$\frac{\{S\}\ \texttt{C}\ \{X'\} \quad \mathit{stab}(X') = R \quad \texttt{C}\text{ is read-only.}}{\{\boxed{S}\}\ \texttt{atomic C}\ \{\boxed{R}\}}$$

## 3 Example: Lock Coupling List

We demonstrate, by example, that our tool can automatically verify the safety of a fine-grained concurrent linked list. We associate one lock per list node rather than have a single lock for the entire list. The list has operations `add` which adds an element to the list, and `remove` which removes an element from the list. Traversing the list uses *lock coupling*: the lock on one node is not released until the next node is locked. Somewhat like a person climbing a rope "hand-over-hand," you always have at least one hand on the rope.

Figure 2 contains the annotated input to our tool. Next, we informally describe the annotations required, and also the symbolic execution of our tool. In the tool the assertions about shared states are enclosed in `[ ... ]` brackets, rather than

```
action Lock(x)      [x|->lk=0,tl=_w  ] [x|->lk=TID,tl=_w]
action Unlock(x)    [x|->lk=TID,tl=_w] [x|->lk=0,tl=_w]
action Add(x,y)     [x|->lk=TID,tl=_w] [x|->lk=TID,tl=y * y|->tl=_w]
action Remove(x,y)  [x|->lk=TID,tl=y * y|->lk=TID,tl=_z] [x|->lk=TID,tl=_z]

ensures: [a!=0 *lseg(a,0)]
init() { a = new(); a->tl = 0; a->lk = 0; }

lock(x) { atomic(x->lk == 0) { x->lk = TID; } as Lock(x); }
unlock(x) { atomic { x->lk = 0; } as Unlock(x); }
```

```
requires: [a!=0 * lseg(a,0)]
ensures:  [a!=0 * lseg(a,0)]
add(e) { local prev,curr,temp;
  prev = a;
  lock(prev);                           (a)
  atomic { curr = prev->tl; }           (b)
  if (curr!=0)
    atomic { temp = curr->hd; }
  while(curr!=0 && temp<e) {
    lock(curr);
    unlock(prev);                       (c)
    prev = curr;
    atomic { curr = prev->tl; }
    if (curr!=0)
      atomic { temp = curr->hd; }
  }
  temp = new();                      ⎫
  temp->lk= 0;                       ⎬  (d)
  temp->hd = e;                      ⎪
  temp->tl = curr;                   ⎭
  atomic { prev->tl = temp; }
         as Add(prev,temp);             (e)
  unlock(prev);
}
```

```
requires: [a!=0 * lseg(a,0)]
ensures:  [a!=0 * lseg(a,0)]
remove(e) { local prev,curr,temp;
  prev = a;
  lock(prev);
  atomic { curr = prev->tl; }
  if (curr!=0)
    atomic { temp = curr->hd; }
  while(curr!=0 && temp!=e) {
    lock(curr);
    unlock(prev);
    prev = curr;
    atomic { curr = prev->tl; }
    if (curr!=0)
      atomic { temp = curr->hd; }
  }
  if (curr!=0) {
    lock(curr);
    atomic { temp = prev->tl; }
    atomic { prev->tl = temp; } ⎫
         as Remove(prev,curr);  ⎬  (f)
    dispose(curr);              ⎭
  }
  unlock(prev);
}
```

**Fig. 2.** Lock-coupling list. Annotations are in italic font.

a box. For example, in the assertion `x|->hd=9 * [y|->hd=10]`, the cell at `x` is local whereas that at `y` is shared.

Note that we calculate loop invariants with a standard fixed-point computation, which uses the same abstraction function as for stabilisation.

We proceed by explaining the highlighted parts of the verification (a)-(f).

*Executing an atomic block (a).* First, we illustrate the execution of an atomic block by considering the first `lock` in the `add` function, following the rule in the previous section. (Step 1) We execute the guard and find the frame.

```
 prev==a * a!=0 * lseg(a,0)
   assume(prev->lk == 0);
 prev==a * a!=0 * prev|->lk:0,tl:_z * lseg(_z,0)
```

The execution unrolls the list segment, because `a!=0` ensures that the list is not empty. Then, we check that the annotated action's precondition holds, namely `prev|->lk=0,tl=_w`. (Any variable starting with an underscore, such as `_w`, is an existential variable quantified across the pre- and post-condition of the action.) The checking procedure computes the leftover formula – the *frame* – obtained by removing cell `prev`. For this atomic block the frame is `lseg(_z,0)`. The frame is not used by the atomic block, and hence remains true at the exit of the atomic block.

Next (Step 2), we execute the body of the atomic block starting with the separate conjunction of the local state and the precondition of the action, so `prev==a * a!=0 * prev|->lk:0,tl:_z * _w=_z` in total. At the end, we get `prev==a * a!=0 * prev|->lk:TID,tl:_z * _w==_z`.

(Step 3) We try to prove that this assertion implies the postcondition of the action plus some local state. In this case, all the memory cells were consumed by the postcondition; hence, when exiting the atomic block, no local state is left.

(Step 4) So far, we have derived the postcondition `[prev|->lk=TID,tl=_z * lseg(_z,0)]`, but we have not finished. We must *stabilise* the postcondition to take into account the effect of other threads onto the resulting state. Following the fixed point computation of Section 2.3, we compute a weaker assertion that is stable under interference from all possible actions of other threads. In this case, the initial assertion was already stable.

*Executing a read-only atomic block (b).* The next atomic block only reads the shared state without updating it. Hence, we require no annotation, as this action causes no interference. Symbolic execution proceeds normally, allowing the code to access the shared state. Again, when we exit the region, we need to stabilise the derived post-condition.

*Stabilisation (c).* Next we illustrate how stabilisation forgets information. Consider unlocking the `prev` node within the loop. Just before unlocking `prev`, we have the shared assertion:

$$\boxed{\mathsf{lseg}(\mathsf{a}, \mathtt{prev}) * \mathtt{prev} \mapsto (lk{=}\mathtt{TID}, tl{=}\mathtt{curr}) * \mathtt{curr} \mapsto (lk{=}\mathtt{TID}, tl{=}\_\mathsf{z}) * \mathsf{lseg}(\_\mathsf{z}, 0)}.$$

This says that the shared state consists of a list-segment from `a` to `prev`, two adjacent locked nodes `prev` and `curr`, and a list segment from `_z` to `nil`. Just after unlocking the node, before stabilisation, we get:

$$\boxed{\mathsf{lseg}(\mathsf{a}, \mathtt{prev}) * \mathtt{prev} \mapsto (lk{=}0, tl{=}\mathtt{curr}) * \mathtt{curr} \mapsto (lk{=}\mathtt{TID}, tl{=}\_\mathsf{z}) * \mathsf{lseg}(\_\mathsf{z}, 0)}.$$

Stabilisation first forgets that $\mathtt{prev} \rightarrow lk = 0$, because another thread could have locked the node; moreover, it forgets that `prev` is allocated, because it could have been deleted by another thread. The resulting stable assertion is:

$$\boxed{\mathsf{lseg}(\mathsf{a}, \mathtt{curr}) * \mathtt{curr} \mapsto (lk{=}\mathtt{TID}, tl{=}\_\mathsf{z}) * \mathsf{lseg}(\_\mathsf{z}, 0)}.$$

*Local updates (d).* Next we illustrate that local updates do not need to consider the shared state. Consider the code after the loop in add. As temp is local, the creation of the new cell and the two field updates affect only the local state. These commands cannot affect the shared state. Additionally, as temp is local state, we know that no other thread can alter it. Therefore, we get the following symbolic execution:

```
[a!=0 * lseg(a,prev) * prev|->lk=TID,tl=curr * lseg(curr,0)]
  temp = new();  temp->lk = 0;  temp->val = e; temp->tl = z;
[a!=0 * lseg(a,prev) * prev|->lk=TID,tl=curr * lseg(curr,0)]
             * temp|->lk=0,val=e,tl=curr
```

*Transfering state from local to shared (e).* Next we illustrate the transfer of state from local ownership to shared ownership. Consider the atomic block with the Add annotation:

```
[a!=0 * lseg(a,prev) * prev|->lk=TID,tl=curr * lseg(curr,0)]
      * temp|->lk=0,tl=curr
  atomic { prev->tl = temp } as Add(prev,temp);
[a!=0 * lseg(a,prev) * prev|->lk=TID,tl=temp
             * temp|->tl=curr * lseg(curr,0)]
```

We execute the body of the atomic block starting with the separate conjunction of the local state and the precondition of the action, so prev|->lk=TID,tl=curr * temp|->lk=0,tl=curr in total. At the end, we get prev|->lk=TID,tl=temp * temp|->lk=0,tl=prev and we try to prove that this implies the postcondition of the action plus some local state. In this case, all the memory cells were consumed by the postcondition; hence, when exiting the atomic block, no local state is left. Hence the cell temp is transferred from local state to shared state.

*Transfering state from shared to local (f).* This illustrates the transfer of state from shared ownership to local ownership, and hence that shared state can safely be disposed. Consider the atomic block with a Remove annotation.

```
[lseg(a,prev) * prev|->lk=TID,tl=curr
          * curr|->lk=TID,tl=temp * lseg(temp,0)]
  atomic {  prev->tl = temp; } as Remove(x,y);
[lseg(a,prev) * prev|->lk=TID,tl=temp * lseg(temp,0)]
          * curr|->lk=TID,tl=temp
```

Removing the action's precondition from the shared state leaves the frame lseg(a,prev) * lseg(temp,0). Executing the body with the action's precondition gives prev|->lk=TID,tl=temp * curr|->lk=TID,tl=temp and we try to prove that this implies the postcondition of the action plus some local state. The action's postcondition requires prev|->lk=TID,tl=temp; so the remaining curr|->lk=TID,tl=temp is returned as local state. This action has taken shared state, accessible by every thread, and made it *local* to a single thread. Importantly, this means that the thread is free to dispose this memory cell, as no other thread will attempt to access it.

| Program | LOC | LOA | Act | #Iter | #Prover calls | Mem(Mb) | Time (sec) |
|---|---|---|---|---|---|---|---|
| lock coupling | 50 | 9 | 4 | 365 | 3879 | 0.47 | 3.9 |
| lazy list | 58 | 16 | 6 | 246 | 8254 | 0.70 | 13.5 |
| optimistic list | 59 | 13 | 5 | 122 | 4468 | 0.47 | 7.1 |
| blocking stack | 36 | 7 | 2 | 30 | 123 | 0.23 | 0.06 |
| Peterson's | 17 | 24 | 10 | 136 | 246 | 0.47 | 1.35 |

**Fig. 3.** Experimental results

```
[lseg(a,x) * x|->lk=TID,tl=z * lseg(z,0)] * y|->lk=TID,tl=z
  dispose(y);
[lseg(a,x) * x|->lk=TID,tl=z * lseg(z,0)]
```

*Summary.* Our example has illustrated fine-grained locking, in particular

- dynamically allocated locks
- non-nested lock/unlock pairs
- disposal of memory (including locks)

Other examples we handle include optimistic reads from shared memory and lazy deletions.

## 4 Experimental Results

Our implementation is based on SmallfootRG, our extension of the separation logic tool called Smallfoot [4]. The tests were executed on a Powerbook G4 1.33 GHz with 786MB memory running OSX 10.4.8. The results are reported in Figure 3. For each example we report: the number of lines of code (LOC) and of annotation (LOA); the number of user-provided actions (Actions); the total number of iterations for all the fixpoint calculations for stabilisation (#Iter); the number of calls to the underlying theorem prover during stabilisation (#Prover calls); the maximum memory allocated during execution (Mem (Mb)), and the total execution time (Time (sec)).

We have tested our tool on a number of fine-grained concurrency examples. The first three (lock coupling, lazy list, optimistic list), taken from [27], all implement the data structure of a set as a singly linked list with a lock per node.

- `lock coupling` The main part of the algorithm was described in Section 3. When traversing the list, locks are acquired and released in a "hand over hand" fashion.
- `lazy list` An algorithm by Heller et al [13], which traverses the list without acquiring any locks; at the end it locks the relevant node and validates the node is still in the list. Deletions happen in two steps: nodes are first marked as deleted, then they are physically removed from the list.
- `optimistic list` Similar to lazy list, it traverses the list without acquiring any locks; at the end it locks the relevant node and re-traverses the list to validate that the node is still in the list.

The next two examples are simpler: `blocking stack` simply acquires a lock before modifying the shared stack; and `peterson` [22] is a well-known mutual exclusion algorithm.

We have a final example of Simpson's `4Slot` [26], which implements a wait-free atomic memory cell with a single reader and a single writer. This algorithm has been verified in both our tool, and Smallfoot. In our new tool it takes under 4 minutes, while in the original Smallfoot it took just under 25 minutes. Also, the specification of the invariant for Smallfoot is over twice as long as the action specification for our tool.[5]

| **Program** | **Lines of Annotation** | **Time (sec)** |
|---|---|---|
| 4Slot (our tool) | 42 | 221 |
| 4Slot (Smallfoot) | 80 | 1448 |

Smallfoot requires the same invariant about shared state at every program point. Our tool calculates all the pertinent shared states at each atomic block, so when it enters an atomic block it does not need to consider as many possibilities as Smallfoot.

Apart from the 4Slot algorithm, we believe our tool takes an acceptable amount of time to verify the algorithms discussed in this section. Our examples have demonstrated the disposal of memory (lock-coupling list and blocking stack), the optimistic reading of values and leaking memory (lazy and optimistic list algorithms), and classic mutual exclusion problems (Peterson's and Simpson's algorithm).

## 5 Conclusion and Related Work

The main challenge for automatic verification of thread-based concurrent programs is finding effective techniques to reason about the interference between threads. Fine-grained concurrency and deep heap updates exacerbate the problems. Several techniques have been proposed to support modular reasoning in the presence of concurrency. These include partial-order reduction [6], thread-modular model checking [11], assume-guarantee reasoning [15], and spatial separation with concurrent separation logic [18,5,21].

Flanagan, Freund, Qadeer and Seshia [9] have a tool, called Calvin, which uses rely/guarantee to reason about concurrent programs. It is built on top of ESC/Java [10]. Unlike our tool, they must check interference on every instruction as they do not have the dynamic partitioning between thread local and shared state.

The Zing [1] model checker has been used to verify some simple fine-grained concurrency examples [24]. The difficulty here is that the environment considered (*i.e.*, the inputs to the program and the actions of the surrounding environment) must be modelled concretely due to the fact that Zing is an *explicit-state* rather

[5] The specification for both could be simplified if Smallfoot directly supported arrays in the assertion language.

than symbolic model checker. In practice this means that the proofs in [24] consider only a limited subset of the potential set of inputs and environment actions.

Yahav and Sagiv [29] use shape analysis to verify a non-blocking queue algorithm. They provide specific instrumentation predicates for the test program in order to guide the automatic analyser. With our approach, the user specifies the atomic actions instead. Their tool does not attempt to decrease the level of interleaving. In principle, it should be possible to combine our technique with their generic analyser with the aim to reduce the number of shapes that they need to consider because of the interference between threads.

Our tool employs a new technique which takes aspects from both assume-guarantee and concurrent separation logic. We are not aware of any other tools doing that. We were also unable to find in the literature case studies of automatic[6] verification tools on a suite of fine-grained concurrent programs. Our work builds on mechanisms developed for program verification with separation logic [3], and the subsequent abstract interpretation techniques for local shape analysis [8].

We have demonstrated that our approach is effective for proving safety and data structure integrity of several list-manipulating algorithms with fine-grained concurrency. In the future, we want to consider other data structures, such as trees and arrays, and to push the barrier towards proving full correctness. We are also interested in inferring automatically the specifications of the actions operating on the shared state, perhaps borrowing from the ideas from thread modular shape analysis [12] or abstraction refinement [14]. More generally, we speculate that the combination of separation logic and rely/guarantee might help producing more effective verification tools for many more classes of concurrent programs.

*Acknowledgements.* We are grateful to the East London Massive for helpful discussions, and to Byron Cook and Alan Mycroft and Peter O'Hearn for comments on earlier drafts of the paper. We gratefully acknowledge the financial support of the EPSRC and Royal Academy of Engineering and a scholarship from the Cambridge Gates Trust.

## References

1. Andrews, T., Qadeer, S., Rajamani, S.K., Xie, Y.: Zing: Exploiting program structure for model checking concurrent software. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 1–15. Springer, Heidelberg (2004)
2. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, Springer, Heidelberg (2004)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, Springer, Heidelberg (2005)
4. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Automatic modular assertion checking with separation logic. In: 4th FMCO, pp. 115–137 (2006)
5. Brookes, S.D.: A semantics for concurrent separation logic (Extended version to appear in *Theoretical Computer Science*). In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 16–34. Springer, Heidelberg (2004)

[6] An interactive proof of full correctness of the lazy list algorithm using PVS was presented in [7].

6. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
7. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set. In: 18th CAV (2006)
8. Distefano, D., O'Hearn, P., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
9. Flanagan, C., Freund, S.N., Qadeer, S., Seshia, S.A.: Modular verification of multithreaded programs. Theor. Comput. Sci. 338(1-3), 153–183 (2005)
10. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI, New York, USA, pp. 234–245. ACM Press, New York (2002)
11. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: SPIN (2003)
12. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: PLDI (2007)
13. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, B., Shavit, N.: A lazy concurrent list-based set algorithm. In: 9th OPODIS (December (2005)
14. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread-modular abstraction refinement. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 262–274. Springer, Heidelberg (2003)
15. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Conference (1983)
16. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
17. Morgan, C.: The specification statement. ACM Trans. Program. Lang. Sys. 10(3), 403–419 (1988)
18. O'Hearn, P.: Resources, concurrency and local reasoning. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 271–307. Springer, Heidelberg (2004)
19. O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: 31st POPL, pp. 268–280 (2004)
20. Owicki, S., Gries, D.: Verifying properties of parallel programs: An axiomatic approach. Comm. ACM 19(5), 279–285 (1976)
21. Parkinson, M., Bornat, R., O'Hearn, P.: Modular verification of a non-blocking stack. In: 34th POPL (2007)
22. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. 12(3), 115–116 (1981)
23. Pnueli, A.: The temporal semantics of concurrent programs. Theoretical Computer Science 13(1), 45–60 (1981)
24. Qadeer, S., Rajamani, S.K., Rehof, J.: Summarizing procedures in concurrent programs. SIGPLAN Not. 39(1), 245–255 (2004)
25. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th LICS, pp. 55–74. IEEE Computer Society Press, Los Alamitos (2002)
26. Simpson, H.: Four-slot fully asynchronous communication mechanism. IEE Proceedings 137(1), 17–30 (1990)
27. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: 11th PPoPP, pp. 129–136. ACM Press, New York (2006)
28. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: 18th CONCUR (2007)
29. Yahav, E., Sagiv, M.: Automatically verifying concurrent queue algorithms. Electr. Notes Theor. Comput. Sci., vol. 89(3) (2003)

# Static Analysis of Dynamic Communication Systems by Partner Abstraction*

Jörg Bauer[1] and Reinhard Wilhelm[2]

[1] Informatics and Mathematical Modelling, Technical University of Denmark,
Kongens Lyngby, Denmark
joba@imm.dtu.dk
[2] Informatik; Univ. des Saarlandes, Saarbrücken, Germany
wilhelm@cs.uni-sb.de

**Abstract.** Prominent examples of dynamic communication systems include traffic control systems and ad hoc networks. They are hard to verify due to inherent unboundedness. Unbounded creation and destruction of objects and a dynamically evolving communication topology are characteristic features.

*Partner graph grammars* are presented as an adequate specification formalism for dynamic communication systems. They are based on the single pushout approach to algebraic graph transformation and specifically tailored to dynamic communication systems. We propose a new verification technique based on abstract interpretation of partner graph grammars. It uses a novel two-layered abstraction, *partner abstraction*, that keeps precise information about objects *and* their communication partners. We identify statically checkable cases for which the abstract interpretation is even complete. In particular, applicability of transformation rules is preserved precisely. The analysis has been implemented in the hiralysis tool. It is evaluated on a complex case study, car platooning, for which many interesting properties can be proven automatically.

## 1 Introduction

We propose a new static analysis for systems with an unbounded number of dynamically created, stateful, linked objects with a constantly evolving communication topology: *dynamic communication systems*. Prominent examples of such systems are wireless communication-based traffic control systems and ad-hoc networks, which have to meet safety-critical requirements that are hard to verify due to the dynamics and unboundedness of dynamic communication systems. A rather obvious key observation will facilitate both the specification and verification of dynamic communication system later on:

**The Partner Principle.** The behavior of an object in a dynamic communication system is determined by its state and by the state of its communication partners.

* This work was supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See www.avacs.org for more information.

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 249–264, 2007.

The partner principle drives both the choice of the specification formalism and the design of the abstraction. Dynamic communication systems are intuitively modeled using graph grammars (or graph transformation systems), because a state of a dynamic communication system can be modeled as a graph, hence the evolution of states as graph grammar rules. Graph grammars come in many flavors, and we refer to [1] for an overview. Our specification formalism, *partner graph grammars*, employs the single pushout approach to graph transformation with a restricted form of (negative) application conditions, called *partner constraints*, to model dynamic communication systems. Partner constraints restrict the applicability of rules and reflect the partner principle.

A partner graph grammar $\mathsf{G}$ consists of a set of rules and an initial graph. The *graph semantics* $[\![\mathsf{G}]\!]$ of $\mathsf{G}$ is the set of all graphs generated by application of rules starting from the initial graph. For dynamic communication systems, there are typically an infinite number of graphs of unbounded size, making verification a hard task.

We aim at automatically computing a bounded over-approximation $[\![\mathsf{G}]\!]^{\alpha}$ of $[\![\mathsf{G}]\!]$. Our technique is based on abstract interpretation [2], where two things need to be defined. First, an *abstraction* $\alpha(G)$ for a single graph $G$; second, *abstract transformers*, *i.e.*, how to apply rules to abstract graphs.

Abstraction of graphs is called *partner abstraction*. As the name suggests, it is again motivated by the partner principle and summarizes objects that are conjectured to behave in the same way, namely, objects in the same state with similar communication partners. As abstract transformers, we define *best abstract transformers* in the spirit of [3]. Though not computable in general, we show it can be done for partner graph grammars using the concept of *materialization*, a restricted form of concretizations.

*Contributions.* We present an intuitive specification formalism for dynamic communication systems and an implementation of our analysis that allows to verify topology properties of them. Using graph grammars to specify and verify dynamic communication systems is, to the best of our knowledge, novel. Using our tool, we analyzed the complex platoon case study (*c.f.*, Section 1.1) that earlier approaches failed to verify. Moreover, the tool proves to be well-suited for system design.

On the theoretical side, we present a static analysis of partner graph grammars, which can handle negative application conditions. It is based on a novel abstraction called partner abstraction. Our analysis is shown sound and even complete for some well-defined cases that occur in practice. In particular, we obtain a result stating the exact preservation of rule applicability by partner abstraction.

*Outline.* First, we present our running example: car platooning. After that, we introduce partner graph grammars as an adequate specification formalism for dynamic communication systems. Section 3 describes the abstract interpretation of partner graph grammars by partner abstraction. Section 4 reports on our implementation and experiments with it, before we comment on related work and conclude.

### 1.1 Case Study: Car Platooning

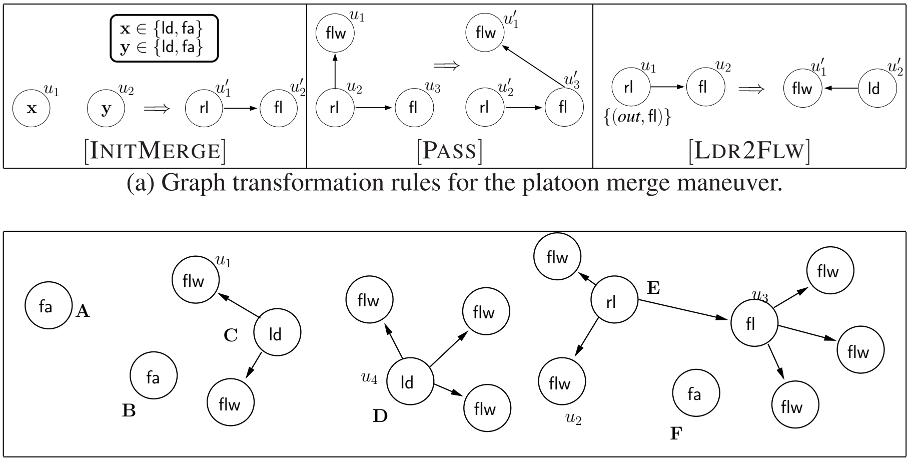
Our case study is taken from the California PATH project [4], the relevant part of which is concerned with cars driving on a highway. In order to make better use of the given space, cars heading for the same direction are supposed to drive very close to each other building *platoons*. Platoons can perform actions like merging or splitting. There are many features that make verification difficult: destruction and dynamic creation of cars, *i.e.*, driving onto and off the highway, an evolving and unbounded communication topology, or concurrency. All the verification methods developed in [4] are inappropriate, because they consider static scenarios with a fixed number of cars only.

A *platoon* consists of a *leader*, the foremost car, along with a number of *followers*. A leader without any followers is called a *free agent* and is considered a special platoon. Within a platoon there are communication channels between the leader and each of its followers. Inter-platoon communication is only between leaders. As a running example, the platoon *merge maneuver* is studied. It allows two approaching platoons to merge. The merge maneuver is initiated by opening a channel between two distinct platoon leaders, *i.e.* leaders or free agents. Then, the rear leader passes its followers one by one to the front leader. Finally, when there are no followers left to the rear leader, it becomes itself a follower to the front leader.

*A Partner Graph Grammar for Platoon Merge.* The merge maneuver is straightforward to model as partner graph grammar with nodes representing cars and edges representing communication channels. Internal states of cars are modeled as node labels. The rules $\mathsf{R}_{\mathrm{merge}}$ of a partner graph grammar modeling the merge maneuver are given in Figure 1. We refer to this figure for an intuitive understanding. The formal notions are introduced in Section 2. We employ five node labels in $\mathsf{R}_{\mathrm{merge}}$. Three of the labels – ld, flw, and fa – represent the states of a car being a leader, follower, or free agent, respectively. Two labels – rl and fl – are used to model situations that occur during a merge maneuver. They distinguish the leader of the rear and the front platoon during a merge. Note that the physical position of platoons is not modeled but abstracted by nondeterminism.

The [InitMerge] rule models the initiation of a merge maneuver. In fact, the rule stated in Figure 1 is a shorthand denoting four rules, one for each possible combination of leaders and free agents. Followers are handed over from the rear to the front leader in rule [Pass]. Eventually, after passing all followers, the rule [Ldr2Flw] can be applied yielding a merged platoon. It makes use of a *partner constraint* requiring the rear leader not to have any followers left. The partner constraint is the set attached to node $u_1$ in rule [Ldr2Flw]. It restricts the application of this rule to cases, where the rear leader has an outgoing edge to a front leader and no other incident edges. There are two simple rules, [Create] and [Destroy], that are not given in Figure 1 but belong to $\mathsf{R}_{\mathrm{merge}}$. The [Create] rule has an empty left graph and a single fa-labeled node as right graph. It caters for unconstrained creation of free agent cars. The [Destroy] rule is the inverse rule, whose application removes a free agent.

Part (b) of Figure 1 shows a sample graph generated by $\mathsf{R}_{\mathrm{merge}}$. Subgraphs **A**, **B**, and **F** are free agent platoons. **C** and **D** are platoons of three and four cars, respectively, whereas **E** depicts a snapshot during a merge maneuver. All these subgraphs are connected components of the graph. As connected components become crucial later on, we shall also call them *clusters*. Cluster **D** may evolve from clusters **B** and **C** by the application of [InitMerge] and a subsequent application of [Ldr2Flw]. For the application of [InitMerge], $\mathbf{x}$ in this rule is set to fa and $\mathbf{y}$ to ld. This represents the case, where the free agent approaches the platoon from behind.



(a) Graph transformation rules for the platoon merge maneuver.

(b) A sample communication topology.

**Fig. 1.** The platoon merge maneuver. Part (a) shows three graph transformation rules: the initiation of a merge, the hand-over of followers from rear to front leader, and the end of a merge. Part (b) shows a graph generated from an empty graph by these rules. Node labels are shown inside nodes.

We are interested in proving topology properties of the platoon merge maneuver. Examples of such properties are: (i) cars have a unique leader, unless they are free agents, or (ii) the asymmetry of the leadership relation. The latter means that there are no two cars considering each other to be their leader.

## 2 Partner Graph Grammars

The purpose of this section is to introduce our formalism for the specification of dynamic communication systems. It is called *partner graph grammars* and is based on the single-pushout approach to graph transformation. Partner graph grammars are restricted to injective matches. They feature a restricted form of negative application conditions called partner constraints intuitively described in Section 1.1. They are an important feature for concise specifications, because they allow to express, when a rule must *not* be applied.

### 2.1 Preliminaries

We recall briefly some standard concepts and notations for reasoning about graphs. Let $\mathcal{L}$ be a finite set of *node labels*. A finite *directed, node-labeled graph*—or graph—$G$ over $\mathcal{L}$ is a triple $(V, E, \mathsf{lab})$, where $V$ is a finite set of nodes, $E \subseteq V \times V$ is a set of edges, and $\mathsf{lab} : V \to \mathcal{L}$ is a labeling mapping. The set of nodes, edges, and the labeling of a graph $G$ are written $V_G$, $E_G$, and $\mathsf{lab}_G$, respectively. The set of all finite graphs over $\mathcal{L}$ is written $\mathcal{G}(\mathcal{L})$. The unique graph without nodes is called the *empty graph* and written $\mathsf{E}$. The disjoint graph union of $G_1$ and $G_2$ is written $G_1 \dot{\cup} G_2$, where $\dot{\cup}$ is also used for disjoint set union. Let $G \in \mathcal{G}(\mathcal{L})$ be a graph and $v \in V_G$ be a node. The set of *incoming partners* of $v$ is defined to be $\rhd^G v = \{w \in V_G \mid (w,v) \in V_E\}$. Analogously, the set $v\rhd^G$ of *outgoing partners* and the set of *partners* $\mathsf{pa}_G(v) = \rhd^G v \cup v\rhd^G$ are defined. Let $G \in \mathcal{G}(\mathcal{L})$ be a graph and $\mathcal{R} \subseteq V_G \times V_G$ an equivalence relation, such that $v_1 \mathcal{R} v_2$ implies $\mathsf{lab}_G(v_1) = \mathsf{lab}_G(v_2)$. The *quotient graph* $G/\mathcal{R}$ is defined to be the graph $H$ with $V_H = V_G/\mathcal{R}$, $E_H = \{([v_1],[v_2]) \mid (v_1,v_2) \in E_G\}$, and $\mathsf{lab}_H = \lambda[v].\mathsf{lab}_G(v)$. Two nodes $v_1, v_2 \in V_G$ are *connected*, iff there exist $u_1, \ldots, u_n \in V_G$, where $u_1 = v_1$, $u_n = v_2$, and $u_i \in \mathsf{pa}_G(u_{i+1})$ for $1 \leq i < n$. Graph $G$ is *connected*, iff all its nodes are pairwise connected. Let $G, H \in \mathcal{G}(\mathcal{L})$ be graphs. A mapping $h : V_G \to V_H$ is called a *morphism*, iff $\mathsf{lab}_G(v) = \mathsf{lab}_H \circ h(v)$ for all $v \in V_G$, and $(h(v_1), h(v_2)) \in E_H$ for all $(v_1, v_2) \in E_G$. Graph $G$ is a *subgraph* of $H$, written $G \leq H$, iff there exists an injective morphism from $G$ to $H$. $G$ and $H$ are isomorphic, written $G \cong H$, if $G \leq H$ and $H \leq G$. A connected graph $G \leq H$ is a *connected component* of $H$, iff $G' = G$ for all connected graphs $G'$ with $G \leq G' \leq H$. The set of all connected components of $H$ is written $\mathsf{cc}(H)$. Often, we use the term *cluster* instead of connected component.

## 2.2 Definition

Let $\mathcal{L}$ be an arbitrary finite set of node labels in the remainder. A partner graph grammar is a pair $(\mathsf{R}, \mathsf{I})$ of a set of *graph transformation rules* and an *initial graph*. Each transformation rule is a four-tuple $(L, h, p, R)$, where $L$ and $R$ are graphs and $h$ maps nodes in $L$ to nodes in $R$ injectively. The mapping $p$ may associate a *partner constraint* with a node in $L$. A rule *matches* another graph $G$, iff $L$ is a subgraph of $G$ due to an injective morphism $m$—called *match*— and if partner constraints are satisfied. For a given node $v$ in $L$, the partner constraint $p(v)$ restricts the possible sets of partners of $m(v)$ in $G$. It can be seen as a mixture of negative and positive application conditions. A partner constraint requires certain partners to be there and, at the same time, all others to be absent. Formally, this shows in the equality required in part 2 of Definition 1.

If a transformation rule $(L, h, p, R)$ matches graph $G$ due to match $m$, it can be *applied to* $G$. The result of the application is the graph $H$, whose nodes are computed as follows: For each node $v \in V_L$ that is not in the domain of $h$, the node $m(v)$ is removed from $G$. On the other hand, each node $v \in V_R$ that is not in the codomain of $h$ is disjointly added to the remaining nodes of $G$, while for each node $v$ in the domain of $h$, the node $m(v)$ remains in $G$. The edges of $H$ are obtained by removing edges from $G$ that are incident to disappearing nodes. Moreover, for all edges $(v_1, v_2)$ in $V_L$, such that both $m(v_1)$ and $m(v_2)$ remain in $V_H$, the edge $(m(v_1), m(v_2))$ is removed from $G$, while all edges in $R$ are added. In Definition 1, $\rightharpoonup$ denotes partial mappings

**Definition 1 (Partner Graph Grammar)**

1. *A* graph transformation rule $r$ *is a four-tuple* $(L, h, p, R)$*, where* $L, R \in \mathcal{G}(\mathcal{L})$*,* $p : V_L \rightharpoonup \wp(\{in, out\} \times \mathcal{L})$*, and* $h : V_L \rightharpoonup V_R$ *is injective. A* partner graph grammar $\mathsf{G}$ *is a pair* $(\mathsf{R}, \mathsf{I})$*, where* $\mathsf{R}$ *is a finite set of graph transformation rules and* $\mathsf{I} \in \mathcal{G}(\mathcal{L})$ *is the* initial graph.
2. *Let* $G \in \mathcal{G}(\mathcal{L})$ *be a graph and* $(L, h, p, R)$ *a graph transformation rule. An injective morphism* $m$ *from* $L$ *to* $G$ *is called a* match*, iff it satisfies partner constraints,* i.e.*, if for all* $v \in dom(p)$*:* $p(v) = \{in\} \times \mathsf{lab}_G(\rhd^G m(v)) \cup \{out\} \times \mathsf{lab}_G(m(v)\rhd^G)$.

3. *If $r = (L, h, p, R)$ matches $G$ by match $m$, the* result of the application *is the graph $H$ obtained as follows:*
   - $V_H = (V_G \setminus m(V_L \setminus \mathit{dom}(h))) \dot{\cup} (V_R \setminus h(\mathit{dom}(h)))$
   - $E_H = (E_G \cap (V_H \times V_H)) \setminus \{(m(u), m(v)) \mid (u, v) \in E_L\} \cup \{(m'(u), m'(v)) \mid (u, v) \in E_R\}$*, where $m' : V_R \to V_H$ such that $m'(v) = m(h^{-1}(v))$, if $v \in h(\mathit{dom}(h))$ and $m'(v) = v$ otherwise.*
   - $\mathsf{lab}_H(v) = \mathsf{lab}_G(v)$*, if $v \notin m(V_L)$; $\mathsf{lab}_H(v) = \mathsf{lab}_R(h(m^{-1}(v)))$, if $v \in m(V_L)$; and $\mathsf{lab}_H(v) = \mathsf{lab}_R(v)$, if $v \notin h(\mathit{dom}(h))$.*

   *In this case, we write $G \rightsquigarrow_r H$.*
4. *Let $\mathsf{G} = (\mathsf{R}, \mathsf{I})$ be a partner graph grammar. For two graphs $G, H \in \mathcal{G}(\mathcal{L})$, we write $G \rightsquigarrow_{\mathsf{R}} H$, iff there exists an $r \in \mathsf{R}$, such that $G \rightsquigarrow_r H$. The* graph semantics $[\![\mathsf{G}]\!]$ *of $\mathsf{G}$ is the set $\{G \in \mathcal{G}(\mathcal{L}) \mid \mathsf{I} \rightsquigarrow_{\mathsf{R}}^* G\}$ of graphs. A sequence $G_1 \rightsquigarrow_{r_1} \ldots \rightsquigarrow_{r_{n-1}} G_n$ is called a* derivation of length $n$.

*Example 1.* The partner graph grammar $\mathsf{G}_{\mathrm{merge}} = (\mathsf{R}_{\mathrm{merge}}, \mathsf{E})$, where $\mathsf{E}$ is the empty graph and $\mathsf{R}_{\mathrm{merge}}$ was specified in Figure 1(a) (except for the simple rules [CREATE] and [DESTROY]), serves as a running example. An element of the graph semantics $[\![\mathsf{G}_{\mathrm{merge}}]\!]$ is shown in Figure 1(b). There, rule [LDR2FLW] does *not* match cluster $\mathbf{E}$, because the partner constraint is not satisfied: The rl-labeled node has both adjacent flw and fl nodes, whereas the partner constraint $\{(\mathit{out}, \mathsf{fl})\}$ requires fl's only. Cluster $\mathbf{D}$ may evolve from $\mathbf{B}$ and $\mathbf{C}$ by an application of [INITMERGE] followed by an application of [LDR2FLW].

Remember the partner principle stated in the introduction. It observes that the behavior of an object is determined by its own state and the state of its communication partners. In terms of partner graph grammars the partner principle is reflected by partner constraints. They allow to define the application conditions of rules in terms of objects and their communication partners. In particular, they can express the *absence* of certain communication partners.

We conclude this section by stating an important property of partner graph grammars with *empty initial graphs*. For every graph $G$ in the graph semantics of a partner graph grammar with an empty initial graph, the disjoint graph union $G \dot{\cup} G$ is also an element of the graph semantics (up to isomorphism). This property is called *graph multiplicity*, and it is often observed in dynamic communication systems. For instance in the platoon case study, it is justified to assume an empty initial highway.

**Lemma 1 (Graph Multiplicity).** *Let $\mathsf{G} = (\mathsf{R}, \mathsf{E})$ be a partner graph grammar. For any graph $G$ holds: If $G \in [\![\mathsf{G}]\!]$, then there exists an $H \in [\![\mathsf{G}]\!]$ with $H \cong G \dot{\cup} G$.*

The proof of the lemma is by induction on the length $n$ of a derivation of $G$ from $\mathsf{E}$. If $n = 1$, then the applied rule must have an empty left graph, because the initial graph is empty. It can thus be applied to $G$, too, yielding $G \dot{\cup} G$ (up to isomorphism). Assume $n > 1$ and a derivation of length $n$: $\mathsf{E} \rightsquigarrow \ldots \rightsquigarrow G' \rightsquigarrow_r G$. By the induction hypothesis, $G' \dot{\cup} G'$ is in $[\![\mathsf{G}]\!]$. As both occurrences of $G$ are not connected, $r$ can be applied to both occurrences independently yielding $G \dot{\cup} G$.

# 3 Partner Abstraction

Our static analysis of dynamic communication systems modeled by partner graph grammars is based on abstract interpretation [2]. Accordingly, we need to define the abstraction of a single graph first. After that, we will say how the application of transformation rules is lifted to abstract graphs. We conclude this section by stating soundness and completeness results.

## 3.1 Abstract Graphs

Abstract graphs are obtained by *partner abstraction*. As the name suggests, this abstraction reflects the partner principle, where we observe the behavior of an object in a dynamic communication system to be determined by its state and the states of its communication partners. It is thus justified to consider two objects *partner equivalent*, if they are in the same state *and* if they have communication partners in the same states.

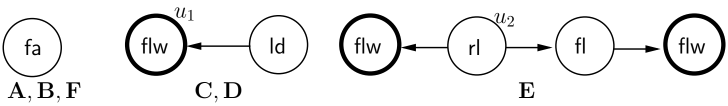
**Definition 2 (Partner Equivalence).** *Let $G \in \mathcal{G}(\mathcal{L})$ be a graph. Nodes $u, v \in V_G$ are* partner equivalent*, written $u \bowtie_G v$, iff* $\mathsf{lab}_G(u) = \mathsf{lab}_G(v)$, $\mathsf{lab}_G(\rhd^G u) = \mathsf{lab}_G(\rhd^G v)$, *and* $\mathsf{lab}_G(u \rhd^G) = \mathsf{lab}_G(v \rhd^G)$. *We denote the equivalence class of $u$* wrt. *partner equivalence by $u \bowtie_G$.*

Note that partner equivalence does not consider the number of adjacent nodes with a given label. It only takes the existence of adjacent nodes and the direction of connecting edges into account. This is a place, where partner abstraction loses some information. It is obvious from Definition 2 that each equivalence class *wrt.* partner equivalence can be uniquely identified by a label and a set of pairs of direction and label. Following [5], we call such characterizing information a *canonical name*.

**Definition 3 (Canonical Names).** *The set* $\mathsf{Names}(\mathcal{L}) = \mathcal{L} \times \wp(\{\mathit{in}, \mathit{out}\} \times \mathcal{L})$ *is called the set of* canonical names over $\mathcal{L}$. *The* canonical name *of node $u$ of graph $G$ is written* $\mathsf{can}_G(u) = (\mathsf{lab}_G(u), \{\mathit{in}\} \times \mathsf{lab}_G(\rhd^G u) \cup \{\mathit{out}\} \times \mathsf{lab}_G(u \rhd^G))$.

The abstraction of graphs is a hierarchical process. First, for each connected component, *i.e.*, for each cluster of a graph, nodes are replaced by their canonical names, which effectively computes the quotient graph *wrt.* partner equivalence cluster-wise. Moreover, we distinguish between singleton equivalence classes and non-singleton equivalence classes. The latter will be called *summary nodes* borrowing terminology from [5]. In general, it is possible to count up to some $k$ serving as a parameter of the abstraction as shown in [6]. For the purpose of this work, however, $k = 1$ suffices.

The second abstraction step is motivated by the partner principle, too. The behavior of an object does not depend on objects, with which it does not communicate. Therefore, in the second abstraction step, we summarize clusters that are isomorphic after quotient graph building. Here, we do not keep any information about the number of summarized clusters. Note that this step summarizes clusters instead of nodes thus yielding a hierarchical abstraction. The notion of canonical naming proves useful for this step, because isomorphic clusters become equal when replacing nodes by their canonical names.



$$u_1 = (\mathsf{flw}, \{(\mathit{in}, \mathsf{ld})\}) \qquad u_2 = (\mathsf{rl}, \{(\mathit{out}, \mathsf{flw}), (\mathit{out}, \mathsf{fl})\})$$

**Fig. 2.** The partner abstraction of the graph of Figure 1(b). Summary nodes are drawn with a thick rim. The clusters that were summarized to one abstract cluster are given below the respective abstract clusters. Two sample node identities, *i.e.*, canonical names, are stated in the bottom line.

**Definition 4 (Abstract Clusters and Graphs).** *Let $C \in \mathcal{G}(\mathcal{L})$ be a connected graph. The* partner abstraction *of $C$ is the pair $(H, \mathsf{mult})$, such that $H$ is a graph with $V_H = \{\mathsf{can}_C(u) \mid u \in V_C\}$, $E_H = \{(\mathsf{can}_C(u), \mathsf{can}_C(v)) \mid (u, v) \in E_C\}$, and $\mathsf{lab}_H = \lambda(\mathsf{a}, P).\mathsf{a}$. The second component is a mapping $\mathsf{mult} : V_H \to \{1, \omega\}$, where $\mathsf{mult}(u) = 1$, if $|u \bowtie_C| = 1$, and $\mathsf{mult}(u) = \omega$, otherwise. We write $\alpha_c(C) = (H, \mathsf{mult})$. The pair $(H, \mathsf{mult})$ is called an* abstract cluster*, and a node $u \in V_H$ with $\mathsf{mult}(u) = \omega$ is called a* summary node. *A set of abstract clusters is called an* abstract graph. *The* partner abstraction *of graph $G \in \mathcal{G}(\mathcal{L})$ is the abstract graph $\alpha(G) = \{\alpha_c(C) \mid C \in \mathsf{cc}(G)\}$.*

*Example 2.* The abstraction of the graph representing a communication topology in Figure 1(b) is presented in Figure 2. Additionally, we show the clusters that are abstracted to the abstract clusters and examples of canonical names of two nodes in the abstract graph.

Some remarks about canonical names are noteworthy:

- All nodes $u$ in abstract cluster $\hat{C} = (C, \mathsf{mult})$ satisfy $\mathsf{can}_C(u) = u$.
- The number of abstract graphs is bounded in terms of the number $l$ of node labels. The maximal number of nodes in an abstract cluster is $n = l \cdot 2^{2l+1}$, *i.e.* there are at most $c = 2^{2n}$ abstract clusters and $2^c$ abstract graphs.
- Partner abstraction constitutes a morphism. The abstraction

  $$\alpha(G) = \{(C_1, \mathsf{mult}_1), \ldots, (C_n, \mathsf{mult}_n)\}$$

  induces a morphism from $G$ to $C_1 \dot\cup \ldots \dot\cup C_n$, that is a mapping from $V_G$ to $V_{C_1} \dot\cup \ldots \dot\cup V_{C_n}$. In the remainder, we shall call it the *induced morphism* $\xi$.

### 3.2 Abstract Transformers

Having defined partner abstraction, the next step is to define abstract transformers, *i.e.*, the application of graph transformation rules to abstract graphs. To do so, we will follow the *best abstract transformer* approach of [3]. Definition 5 first defines the notion of an *abstract match*. It resembles the notion of a match as defined in Definition 1 except for injectivity, which is lost due to summarization of nodes.

If a graph transformation rule $r$ matches an abstract graph $\hat{G}$, we apply it to all concretizations of $\hat{G}$, where a concretization of $\hat{G}$ is a graph $G$ with $\alpha(G) = \hat{G}$. After that, the resulting graphs will be abstracted again using $\alpha$. In general, there may be

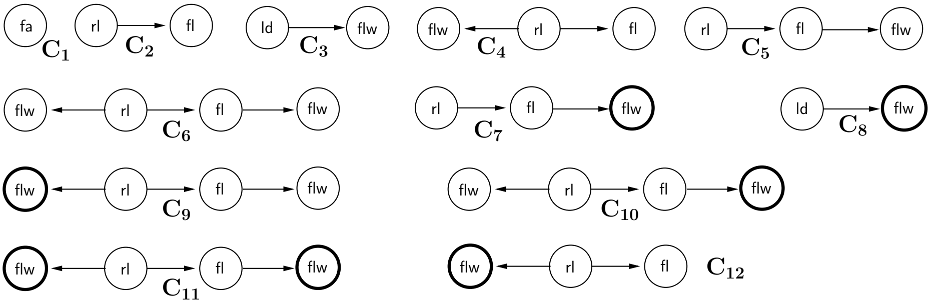


**Fig. 3.** The abstract graph semantics of the platoon merge partner graph grammar $\mathsf{G}_{\mathrm{merge}}$. It consists of 12 abstract clusters.

infinitely many concretizations of $\hat{G}$. Therefore, we identify a subset of all concretizations guaranteed to be finite for any abstract graph $\hat{G}$. Any such concretization is called a *materialization* as defined in Definition 6. In Lemma 2, we show that materializations define the same abstract transformers as concretizations. Hence, they are a way to compute best abstract transformers.

**Definition 5 (Abstract Graph Semantics).** *Let* $\mathsf{G} = (\mathsf{R}, \mathsf{I})$ *be a partner graph grammar, let* $r = (L, h, p, R) \in \mathsf{R}$ *be a graph transformation rule, and let* $\hat{G} = \{(C_1, \mathsf{mult}_1), \ldots, (C_n, \mathsf{mult}_n)\}$ *be an abstract graph.*

1. *A morphism* $m$ *from* $L$ *to* $G := C_1 \dot{\cup} \ldots \dot{\cup} C_n$ *is called an* abstract match *from* $r$ *to* $\hat{G}$*, iff for all* $D \in \mathsf{cc}(L)$ *and for all* $u \in V_D$ *such that* $m(u) \in V_{C_i}$ *holds: If* $u \in dom(p)$*, then* $p(u) = \{in\} \times \mathsf{lab}_{C_i}(\rhd^{C_i} m(u)) \cup \{out\} \times \mathsf{lab}_{C_i}(m(u) \rhd^{C_i})$*; and* $|m^{-1}(m(u)) \cap V_D| > 1$ *implies* $\mathsf{mult}_i(m(u)) = \omega$*.*
2. *Let* $\hat{H}$ *be an abstract graph. It is the* result of an application of $r$ to $\hat{G}$*, written* $\hat{G} \rightsquigarrow_r^\alpha \hat{H}$*, iff there exists an abstract match from* $L$ *to* $\hat{G}$ *and there exist graphs* $M$ *and* $M'$*, such that* $\alpha(M) = \hat{G}$*,* $M \rightsquigarrow_r M'$*, and* $\alpha(M') = \hat{H}$*.*
3. *The* abstract graph semantics $[\![\mathsf{G}]\!]^\alpha$ *of* $\mathsf{G}$ *is defined inductively as* $[\![\mathsf{G}]\!]_0^\alpha = \alpha(\mathsf{I})$*,* $[\![\mathsf{G}]\!]_{i+1}^\alpha = [\![\mathsf{G}]\!]_i^\alpha \cup \bigcup\{\hat{H} \mid \exists \hat{G} \in [\![\mathsf{G}]\!]_i^\alpha, r \in \mathsf{R}. \hat{G} \rightsquigarrow_r^\alpha \hat{H}\}$*, and* $[\![\mathsf{G}]\!]^\alpha = \bigcup_{i \geq 0} [\![\mathsf{G}]\!]_i^\alpha$*.*

Besides potential non-injectivity of an abstract match, there is an additional requirement regarding summary nodes. If more than one node of the same connected component of the left graph of a rule match the same node in an abstract graph, this node must be a summary node. The application of a matching rule is defined in terms of applying the rule to concretizations. Although this definition is not constructive, we will show how to overcome this by using materializations. The abstract graph semantics collects all abstract clusters obtained by iterated rule applications.

*Example 3.* The abstract graph semantics $[\![\mathsf{G}_{\mathrm{merge}}]\!]^\alpha$ of the platoon merge implementation is shown in Figure 3. Sample abstract rule applications are

$$(1)\text{: } \{\mathbf{C_8}\} \rightsquigarrow^\alpha_{[\textsc{InitMerge}]} \{\mathbf{C_{11}}\} \quad (2)\text{: } \{\mathbf{C_9}\} \rightsquigarrow^\alpha_{[\textsc{Pass}]} \{\mathbf{C_{10}}\}$$

In terms of platoons, a concretization justifying (1) contains two platoons of three cars each that merge. They are abstracted to abstract $\{\mathbf{C_8}\}$. The result of the application of [InitMerge] is abstracted to $\{\mathbf{C_{11}}\}$.

We shall now derive upper bounds for the number of concretizations needed to compute abstract transformers. Concretizations within these bounds are called *materializations*. The numbers given in Definition 6 are not always needed, because there are many special cases, where smaller numbers suffice. This is exploited in the implementation of the analysis. In the definition, we use further notations: Given an abstract match $m$ and a node $u$ of abstract cluster $(C, \mathsf{mult})$, $\mathsf{env}(u)$ denotes the number of matched partners of $u$, *i.e.*, $\mathsf{env}(u) = |\,\mathsf{pa}_C(u) \cap \mathit{codom}(m)\,|$. For an abstraction $\alpha(G) = \hat{G}$ with induced morphism $\xi$ and some node $u$ of $\hat{G}$, we write $\mathsf{mater}(u)$ for the number of nodes mapped to $u$ by $\xi$, *i.e.*, $\mathsf{mater}(u) = |\,\xi^{-1}(u)\,|$. We call nodes mapped to $u$ by $\xi$, *nodes materialized from* $u$. Finally, $\mathsf{matched}(u)$ denotes the number of nodes matching node $u$ of an abstract graph, *i.e.*, for an abstract match $m$, $\mathsf{matched}(u) = |\,m^{-1}(u)\,|$. Note that the size of all these entities is statically bounded in terms of the shape of left graphs and the number of node labels.

**Definition 6 (Materialization).** *Let* $\hat{G} = \{(C_1, \mathsf{mult}_1), \ldots, (C_n, \mathsf{mult}_n)\}$ *be an abstract graph and let* $r = (L, h, p, R)$ *be a graph transformation rule such that* $r$ *matches* $\hat{G}$ *with abstract match* $m$*. A concretization* $G$ *of* $\hat{G}$ *is called a* materialization *with respect to* $r$ *and* $m$*, iff* $|\,\mathsf{cc}(G)\,| \leq |\,\mathsf{cc}(L)\,| + n$*, and for each* $C_i$ *and all summary nodes* $u \in V_{C_i}$*:* $\max\{2, \mathsf{matched}(u)\} \leq \mathsf{mater}(u)$ *and* $\mathsf{mater}(u) \leq \mathsf{matched}(u) + 2^{\mathsf{env}(u)+1}$.

**Lemma 2 (Materialization).** *If* $\hat{G} \rightsquigarrow^{\alpha}_{r} \hat{H}$ *with abstract match* $m$*, then there exists a materialization* $M$ *of* $\hat{G}$ wrt. $r$ *and* $m$ *and graph* $M'$ *s.t.* $M \rightsquigarrow_r M'$ *and* $\alpha(M') = \hat{H}$.

The proof of the lemma is based on the observation that nodes in a graph that are not adjacent to a matched node cannot be affected by a rule application. Only matched nodes may change their label and adjacent edges, *i.e.*, only matched nodes *or their partners* may change their canonical name. In any case, at least two or the number of nodes matching a summary node must be materialized from it. If a summary node $u$ is adjacent to $\mathsf{env}(u)$ matched nodes, $2^{\mathsf{env}(u)}$ cases must be distinguished, because a node materialized from $u$ may or may not be connected to each of the matched partners. Hence, it may or may not be affected by the update. The additional factor of 2 in the upper bound is needed because either one or more than one materialized nodes may be affected in each of the $2^{\mathsf{env}(u)}$ ways. If a summary node $u$ is matched and has matched partners, the number of nodes matching $u$ must be materialized, too, yielding the additive in the upper bound.

The bound on the number of clusters in a materialization results from the observation that at most $|\mathsf{cc}(L)|$ clusters can be affected by a transformation rule using $L$. Since there are $n$ abstract clusters, and each needs to be represented in any concretization, the bound for the number of connected components in a materialization is $|\mathsf{cc}(L)| + n$.

Due to the finiteness of all entities bounding the size of materializations also the number of materializations is finite. Lemma 2 also shows that the abstract graph semantics can be computed iteratively in finite time for any partner graph grammar: Since $[\![\cdot]\!]^{\alpha}_{i}$ is monotone in $i$ and since there are only finitely many abstract graphs and finitely many materializations of them the computation of the abstract graph semantics will terminate. Besides termination, we obtain the following soundness result.

**Theorem 1 (Soundness).** *Let* $\mathsf{G}$ *be a partner graph grammar. Then the abstract graph semantics is a sound over-approximation of the graph semantics:* $\bigcup_{G \in [\![\mathsf{G}]\!]} \alpha(G) \subseteq [\![\mathsf{G}]\!]^\alpha$.

The proof is obvious, because we defined abstract updates in terms of best abstract transformers. Theorem 1 is typically used in its counterpositive form to prove properties of partner graph grammars, *i.e.*, of dynamic communication systems. If a graph does not occur as a subgraph of an abstract cluster in the abstract graph semantics, it cannot occur in the concrete graph semantics. In our running example, we can thus prove the asymmetry and the uniqueness of the leadership relation.

In fact, Theorem 1 can be used to prove even stronger properties. Assume a transformation rule, where the right graph is a singleton node with an otherwise unused label. If this label does not occur in the abstract graph semantics, it is guaranteed by soundness that this rule never matches in the concrete graph semantics. The additional strength is gained because there may be partner constraints used in the rule.

### 3.3 Completeness Issues

This section augments the abstract interpretation of partner graph grammars with some unexpected completeness results: First, we identify cases, where partner abstraction preserves precisely the applicability of rules. Second, we present statically checkable criteria that are sufficient for an abstract graph semantics not to contain spurious clusters or even to decide the word problem for a class of partner graph grammars.

It is clear that general completeness results cannot be expected: There is only a bounded number of abstract graphs, whereas the structures in left graphs of transformation rules are unrestricted. However, if we exclude some pathological cases, we can obtain completeness properties.

One pathological case is abstract clusters that include subgraphs like $\mathsf{b} \leftarrow \mathsf{a} \rightarrow \mathsf{b}$, where the $\mathsf{a}$-labeled node is a summary node. Even though we know that all nodes represented by it have an outgoing edge to a $\mathsf{b}$-labeled node, we do not know to which of the two. If such patterns do not occur, we speak of *unique partners*. Furthermore, edges between summary nodes are a source of information loss. For example, the following cycle between summary nodes may result from abstracting a cycle of any even length of alternating $\mathsf{a}$ and $\mathsf{b}$-labeled nodes: $\mathsf{a} \rightleftarrows \mathsf{b}$.

**Definition 7 (Special Graphs).** *Let* $(C, \mathsf{mult})$ *be an abstract cluster. It has* unique partners*, iff for all* $u \in V_C$ *and for all* $\mathsf{a} \in \mathcal{L}$ *both* $|\rhd^C u \cap \mathsf{lab}_C^{-1}(\mathsf{a})|$ *and* $|u \rhd^C \cap \mathsf{lab}_C^{-1}(\mathsf{a})|$ *are at most 1. It has a* summary cycle*, iff there exists an* $n > 1$ *and a subgraph of* $C$ *made up of* $n$ *distinct summary nodes and at least* $n$ *distinct edges among them.*

The definition of summary cycles may seem awkward. It is justified, however, because we are really interested in cycles, where the direction of the edges does not matter (see the proof of Theorem 2 for details).

It is obvious that partner abstraction preserves the applicability of rules, *i.e.*, if a rule matches $G$, it also matches $\alpha(G)$. Without partner constraints, this holds for any homomorphic abstraction, whereas partner abstraction additionally guarantees the preservation of partner constraint satisfaction. The inverse direction, *i.e.* if a rule matches $\alpha(G)$

it also matches $G$, does not hold in general. Consider the example of a cycle between two summary nodes above. This abstract cluster is matched by a rule with a left graph being a cycle of length 8 of alternating $\mathsf{a}$ and $\mathsf{b}$-labeled nodes. However, this rule does not match the concretization being a cycle of length 6. Theorem 2 describes cases when this direction holds. For simplicity, it is formulated in terms of abstract clusters.

**Theorem 2 (Match).** *Let $r = (L, h, p, R)$ be a transformation rule, where $L$ is connected, and let $\hat{C} = (C, \mathsf{mult})$ be an abstract cluster. If there is an injective abstract match from $L$ to $\{\hat{C}\}$, and $\hat{C}$ has unique partners and no summary cycles, then $r$ matches all $D$ with $\alpha_c(D) = \hat{C}$.*

The proof exploits two key observations. First, unique partners imply the following. If $(u, v) \in E_C$, and $\mathsf{mult}(u) = 1$ or $\mathsf{mult}(v) = 1$, then for all $u' \in \xi^{-1}(u)$ and all $v' \in \xi^{-1}(v)$, also $(u', v') \in E_D$. W.l.o.g, assume $\mathsf{mult}(v) = 1$. The observation holds, because we know that all $u'$ must have an outgoing edge to a $\mathsf{lab}_C(v)$-labeled node. As there is only one such partner in $D$, $u'$ must be connected to $v'$. This results implies that abstract matches that do not contain edges between summary nodes can be mimicked in any concretization. For edges between summary nodes, we use the second key observation: As there are no summary cycles, we can organize matched summary nodes in a forest. The concrete match of $L$ to $D$ is then constructed by traversing this forest. If summary node $u$ is the root of a tree, we choose an arbitrary $u' \in \xi^{-1}(u)$ for the concrete match. If $u$ is not a root, it has only one edge to an ancestor in the traversal. Due to the unique partner property, we can then always pick one $u'$ in $D$ that is connected to the pick we made for the ancestor.

We will now define the notion of *complete clusters* and will show in Theorem 3 that complete clusters imply completeness of the abstract interpretation of partner graph grammars. Since we need to apply the cluster multiplicity property stated in Lemma 1, we concentrate on partner graph grammars with an empty initial graph. We shall call a rule with an empty left graph a *create rule*, because the right graph may be created in an unconstrained way. The most intricate notion we need in Definition 8 is the notion of an *inductive summary node*. A summary node $u$ of abstract cluster $\hat{C}_0 = (C, \mathsf{mult}) \in [\![\mathsf{G}]\!]^\alpha$ is inductive, iff there exists $\hat{C}_n = (C, \mathsf{mult}') \in [\![\mathsf{G}]\!]^\alpha$, where $\mathsf{mult}'(u) = 1$. Moreover, there need to be a set $\hat{C}_i = (C_i, \mathsf{mult}_i)$ of abstract clusters, such that $u \in V_{C_i}$ for all $0 < i < n$, and $\hat{C}_{i+1}$ results from a rule application that does not match $u$ from $\hat{C}_i$. Finally, exactly once, one additional node must become partner equivalent to $u$ by a rule application on this path. Informally, an inductive summary node is part of a loop incrementing its size by 1.

*Example 4.* The summary node of abstract cluster $\mathbf{C_8}$ of Figure 3 is inductive with clusters $\mathbf{C_3}$ and $\mathbf{C_5}$. In terms of platoons, we obtain arbitrarily many followers by constantly merging with a free agent.

**Definition 8 (Complete Clusters).** *Let $\mathsf{G} = (\mathsf{R}, \mathsf{E})$ be a partner graph grammar. Abstract cluster $\hat{C} \in [\![\mathsf{G}]\!]^\alpha$ is* complete*, iff one of the following conditions holds:*

1. *There exists a create rule $(\mathsf{E}, h, p, R)$, such that $\alpha_C(R) = \hat{C}$ and $\hat{C}$ does not contain any summary nodes.*

2. *$\hat{C}$ contains exactly one summary node $u$, such that $u$ is inductive with complete clusters without summary nodes.*
3. *$\hat{C}$ results from a rule application to complete abstract clusters, such that no summary nodes are matched in the application or created by the application.*

Proving the completeness of the clusters of an abstract graph semantics amounts to imposing a strict order on the clusters, called a *generating order*. Minimal elements are those that result from the application of a create rule. It should be admitted, however, that we do not have an efficient algorithm to compute a generating order or to prove the existence of one. In some cases, the order can be found by manual inspection:

*Example 5.* All clusters of $[\![\mathsf{G}_{\mathrm{merge}}]\!]^{\alpha}$ are complete. Case 1 applies to $\mathbf{C_1}$. Clusters $\mathbf{C_2}$ and $\mathbf{C_3}$ come next in the order and are proven complete by case 3. The same goes for $\mathbf{C_4}$, $\mathbf{C_5}$, and $\mathbf{C_6}$. As mentioned above, case 2 applies to cluster $\mathbf{C_8}$. All remaining clusters result from applying [InitMerge] to $\mathbf{C_8}$ using case 3.

**Theorem 3 (Completeness).** *Let $\mathsf{G} = (\mathsf{R}, \mathsf{E})$ be a partner graph grammar. If all $\hat{C} \in [\![\mathsf{G}]\!]^{\alpha}$ are complete, then $\bigcup_{G \in [\![\mathsf{G}]\!]} \alpha(G) = [\![\mathsf{G}]\!]^{\alpha}$. If, additionally, $R$ is connected for all $(L, h, p, R) \in \mathsf{R}$, then $G \in [\![\mathsf{G}]\!] \Leftrightarrow \alpha(G) \in [\![\mathsf{G}]\!]^{\alpha}$ (up to isomorphism).*

The proof of the theorem is only sketched here (see [6] for all the details). It is by well-founded induction on the generating order of the clusters and mimics abstract derivations in the concrete graph semantics. First, we need to show the uniqueness—up to isomorphism and number of nodes materialized from summary nodes—of materializations. Then, Theorem 2 and Lemma 1 guarantee that the loop in the definition of inductive summary nodes can be executed arbitrarily many times, always incrementing the actual size of the summary node by 1. All other summary nodes evolve from inductive summary nodes without change of size as ensured by case 3 of Definition 8. Eventually, Lemma 1 together with connected right graphs guarantees that each abstract cluster individually can have an arbitrary number of concretizations independently of other clusters.

For our running example, Theorem 3 implies that we precisely know all the graphs generated by the platoon merge partner graph grammar, because all right graphs of rules in $\mathsf{R}_{\mathrm{merge}}$ are connected. For this particular grammar, we can thus decide the word problem using Theorem 3.

## 4 Experimental Evaluation

We have implemented the abstract interpretation of partner graph grammars in the `hiralysis` tool. It takes as input a textual representation of a partner graph grammar and produces as output a graphical representation of the abstract graph semantics. In addition to the material presented here, `hiralysis` supports edge labels.

The `hiralysis` tool has been evaluated on the platoon case study. Some numerical results are shown in Table 1. It shows the size of the input in terms of numbers of node and edge labels, number of transformation rules, and number of partner constraints used in the partner graph grammar. Finally, the size of the abstract graph semantics in terms of numbers of abstract clusters is given.

**Table 1.** Experiments conducted with the `hiralysis` implementation of our analysis

| | **#rules** | **#node labels** | **#edge labels** | **#pconstraints** | **#abstract clusters** |
|---|---|---|---|---|---|
| **merge** | 8 | 5 | 1 | 1 | 12 |
| **split** | 4 | 6 | 1 | 4 | 13 |
| **combined** | 12 | 6 | 1 | 5 | 169 |
| **combined+** | 12 | 6 | 1 | 9 | 22 |
| **queues** | 30 | 18 | 4 | 34 | 159 |
| **faulty** | 32 | 5 | 1 | 1 | 20 |

Among the examples, our running example, **merge**, is the simplest. Its abstract graph semantics consisting of 12 clusters was explicitly given in Figure 3. Grammar **split** implements a split maneuver, while the two versions of **combined** combine merging and splitting. Note that **combine** simply takes the union of **merge** and **split**. The larger number of abstract clusters results from interferences, where platoons try to merge and split simultaneously. This is a source of mistakes not taken into account in the original PATH project [4]. Grammar **combined+** repairs these mistakes by introducing additional partner constraints that restrict the parallelism: another hint to the importance of partner constraints. For the repaired protocol, we were again able to verify properties like unique leaders.

Grammar **queues** extends the original PATH specification by introducing buffered communication. It is an example where quite complicated graphs are handled by our technique [6]. Grammar **faulty** augments the simple merge maneuver with non-deterministically disappearing, unreliable communication links. Such a model is helpful in discovering potential failure situations. More examples are reported on in [6].

In the experiments conducted so far, run time was not found to be a problem. All abstract semantics' of Table 1 were computed in fractions of seconds. Even thousands of clusters could be handled within few seconds. However, we only encountered such numbers for grammars, where we had introduced flaws unintentionally. Therefore, they do not occur among the results given. Interestingly, the `hiralysis` output proved extremely valuable for debugging of such grammars because of its graphical output.

Admittedly, the experiments so far are of modest size, because they are all implemented by hand. We are currently exploring the automated generation of `hiralysis` input from more fine-grained specifications of dynamic communication systems such as those presented in [7]. The expected partner graph grammars may consist of thousands of rules and will give more hints to the scalability of the technique.

## 5 Related Work

Partner abstraction was inspired by canonical abstraction of [5], which is particularly well-suited to reason about dynamically allocated data structures. There, reachability is crucial, which cannot be expressed by partner abstraction making it a bad choice for analyzing list-like graphs. However, canonical abstraction is flat, that is, it summarizes only nodes. Partner abstraction is two-layered: it summarizes nodes in a first step and clusters in a second step. Moreover, it works fully automatic without intricate instrumentation predicates. Finally, it is tailored to dynamic communication systems, where graph transformation rules seem easier and more intuitive to write down than the predicate update formulas of [5].

The area of verification of concurrent parameterized systems is very active. The two most relevant and most recent approaches are [8] and [9]. While these techniques are able to deal with infinite-state systems in general, they are typically concerned with unbounded data domains rather than unbounded evolving communication topologies rendering them orthogonal. Even if they are interested in communication topologies, it seems that our approach is the only one able to handle completely arbitrary graphs. Communication topologies are the focus of an abstract interpretation of the $\pi$-calculus [10]. However, we believe that $\pi$ is too fine-grained to model topology evolution hence requiring encoding of features that can be modeled directly using graph grammars.

An approach to formal verification of graph grammars [11] proved very successful, but is not based on abstract interpretation. Rather it is based on the unfolding semantics of the given graph grammar [12] and approximates behavior by means of Petri nets. It is mostly targeted to the analysis of heap-manipulating programs.

A completly different application of graph grammar verification (term graph rewriting, to be more precise) comes from the world of functional programming and constitutes the formal safety proof of the strictness analysis employed in the Clean compiler [13]. It is based on abstract interpretation but tailored to this particular safety proof.

Another abstract interpretation based approach was developed independently in [14] and used for software engineering purposes. The underlying abstraction relies on counting incoming and outgoing labeled edges. However, our approach provides clear advantages over [14]: we are not restricted to deterministic graphs; we have an implementation; we have a hierarchical abstraction tailored to dynamic communication systems; we have completeness results. Most importantly, we support (negative) application conditions in terms of partner constraints. Without this feature graph grammars are hardly usable for dynamic communication systems.

## 6 Conclusion

We have presented partner graph grammars as an adequate specification formalism for dynamic communication systems, for which we have defined an abstract interpretation based on a novel, hierarchical abstraction called partner abstraction. The analysis was shown to be sound and, in some well-defined cases, complete. We have reported on the `hiralysis` implementation of the analysis that has been evaluated on the complex platoon case study – automatically revealing flaws and proving so far unproven topology properties.

We are currently working on getting rid of some of the limitations. Due to the particular choice of partner abstraction, the applicability is mainly limited to dynamic communication systems. The properties we are able to establish are mere properties of graphs rather than temporal properties of graph transition systems.

We are also extending our work on unreliable communication links towards probabilistic models, where links do not fail arbitrarily but as determined by a probability distribution. Finally, we are implementing the automatic generation of `hiralysis` input from more fine-grained models of dynamic communication systems such as those of [7].

## References

1. Rozenberg, G.(ed.).: Handbook of Graph Grammars and Computing by Graph Transformations, vol 1: Foundations. World Scientific, Singapore (1997)
2. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In: Symp. on Princ. of Prog. Lang., New York, NY, pp. 238–252. ACM Press, New York (1977)
3. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Symp. on Princ. of Prog. Lang., pp. 269–282. ACM Press, New York (1979)
4. Hsu, A., Eskafi, F., Sachs, S., Varaiya, P.: The design of platoon maneuver protocols for IVHS. Technical Report UCB-ITS-PRR-91-6, University of California, Berkley (1991)
5. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24(3), 217–298 (2002)
6. Bauer, J.: Analysis of Communication Topologies by Partner Abstraction. PhD thesis, Universität des Saarlandes,(2006) Available from `http://www2.imm.dtu.dk/~joba/phd.pdf`
7. Bauer, J., Schaefer, I., Toben, T., Westphal, B.: Specification and verification of dynamic communication systems. In: Proc. of the 6th Conference on Application of Concurrency to System Design (ACSD 2006), IEEE Computer Society Press, Los Alamitos (2006)
8. Bouajjani, A., Jurski, Y., Sighireanu, M.: A generic framework for reasoning about dynamic networks of infinite-state processes. In: Grumberg, O., Huth, M. (eds.) 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (January 2007)
9. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized verification of infinite-state processes with global conditions. In: Damm, W., Hermanns, H. (eds.) Proc. CAV'07, 19th International Conference on Computer Aided Verification (July 2007)
10. Venet, A.: Automatic determination of communication topologies in mobile systems. In: Static Analysis Symposium, pp. 152–167 (1998)
11. Baldan, P., Corradini, A., König, B.: Verifying finite-state graph grammars: An unfolding-based approach. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 83–98. Springer, Heidelberg (2004)
12. Baldan, P., Corradini, A., Montanari, U.: Unfolding and event structure semantics for graph grammars. In: Thomas, W. (ed.) ETAPS 1999 and FOSSACS 1999. LNCS, vol. 1578, pp. 73–89. Springer, Heidelberg (1999)
13. Clark, D., Hankin, C., Hunt, S.: Safety of strictness analysis via term graph rewriting. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 95–114. Springer, Heidelberg (2000)
14. Rensink, A., Distefano, D.: Abstract graph transformation. Electr. Notes Theor. Comput. Sci. 157(1), 39–59 (2006)

# Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis

Ben Hardekopf and Calvin Lin

The University of Texas at Austin, Austin TX 78712, USA
{benh,lin}@cs.utexas.edu

**Abstract.** Pointer information is a prerequisite for most program analyses, and inclusion-based, *i.e.* Andersen-style, pointer analysis is widely used to compute such information. However, current inclusion-based analyses can have prohibitive costs in time and space, especially for programs with millions of lines of code. We present a suite of offline optimizations that exploit pointer and location equivalence to shrink the input to the subsequent pointer analysis without affecting precision, dramatically reducing both analysis time and memory consumption. Using a suite of six open-source C programs ranging in size from 169K to 2.17M LOC, we demonstrate that our techniques on average improve analysis time by 1.3–2.7× and reduce memory consumption by 3.2–6.9× over the best current techniques.

## 1 Introduction

Most program analyses require pointer information, from traditional compiler optimizations to software verification, security analysis, and program understanding. Many of these analyses are interprocedural and require a highly scalable whole-program pointer analysis to compute pointer information. The precision of the computed information can have a profound impact on the usefulness of the subsequent program analysis. Inclusion-based, *i.e.* Andersen-style, pointer analysis is widely-used because of its relative precision and potential for scalability. Inclusion-based analysis scales to millions of lines of code, but memory consumption is prohibitively high [6]. Memory consumption can be greatly reduced by using BDDs to represent points-to sets, but this significantly increases analysis time [6]. Our goal is to break this trade-off by reducing both memory consumption and analysis time for inclusion-based pointer analysis, without affecting the precision of the results.

Inclusion-based analysis is the most precise flow- and context-insensitive pointer analysis. It extracts inclusion constraints from the program code to approximate points-to relations between variables, representing the constraints using a *constraint graph*, with nodes to represent each program variable and edges to represent the constraints between variables. Indirect constraints—those that involve pointer dereferences—can't be directly represented in the graph, since points-to information isn't available until after the analysis has completed. The analysis satisfies the constraints by computing the dynamic transitive closure of

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 265–280, 2007.

the graph; as new points-to information becomes available, new edges are added to the graph to represent the indirect constraints. The transitive closure of the final graph yields the points-to solution.

Inclusion-based analysis has a complexity of $O(n^3)$ time and $O(n^2)$ space, where $n$ is the number of variables; the key to scaling the analysis is to reduce the input size—*i.e.* make $n$ smaller—while ensuring that precision is not affected. This goal is accomplished by detecting equivalences among the program variables and collapsing together equivalent variables. Existing algorithms only recognize a single type of equivalence, which we call *pointer equivalence*: program variables are pointer equivalent iff their points-to sets are identical. There are several existing methods for exploiting pointer equivalence. The primary method is *online cycle detection* [5,6,7,10,11]. Rountev *et al.* [12] introduce another method called *Offline Variable Substitution* (OVS). An offline analysis is a static analysis performed prior to the actual pointer analysis; in this case, OVS identifies and collapses a subset of the pointer equivalent variables before feeding the constraints to the pointer analysis.

In this paper, we introduce a suite of new offline optimizations for inclusion-based pointer analysis that go far beyond OVS in finding pointer equivalences. We also introduce and exploit a second notion of equivalence called *location equivalence*: program variables are location equivalent iff they always belong to the same points-to sets, *i.e.* any points-to set containing one must also contain the other. Our new optimizations are the first to exploit location equivalence to reduce the size of the variables' points-to sets without affecting precision. Together, these offline optimizations dramatically reduce both the time and memory consumption of subsequent inclusion-based pointer analysis. This paper presents the following major results:

- Using three different inclusion-based pointer analysis algorithms [7,10,6], we demonstrate that our optimizations on average reduce analysis time by 1.3–2.7× and reduce memory consumption by 3.2–6.9×.
- We experiment with two different data structures to represent points-to sets: (1) sparse bitmaps, as currently used in the GCC compiler, and (2) a BDD-based representation. While past work has found that the bitmap representation is 2× faster but uses 5.5× more memory than the BDD representation [6], we find that, due to our offline optimizations, the bitmap representation is on average 1.3× faster and uses 1.7× *less* memory than the BDD representation.

This paper makes the following conceptual contributions:

- We present Hash-based Value Numbering (HVN), an offline optimization which adapts a classic compiler optimization [3] to find and exploit pointer equivalences.
- We present HRU (**H**VN with de**R**eference and **U**nion), an extension of HVN that finds additional pointer equivalences by interpreting both union and dereference operators in the constraints.

- We present LE (**L**ocation **E**quivalence), an offline optimization that finds and exploits location equivalences to reduce variables' points-to set sizes without affecting precision.

## 2 Related Work

Andersen introduces inclusion-based pointer analysis in his Ph.D. thesis [1], where he formulates the problem in terms of type theory. Andersen's algorithm solves the inclusion constraints in a fairly naive manner by repeatedly iterating through a constraint vector.

The first use of pointer equivalence to optimize inclusion-based analysis comes from Faehndrich *et al.* [5], who represent constraints using a graph and then derive points-to information by computing the dynamic transitive closure of that graph. The key optimization is a method for partial online cycle detection.

Later algorithms expand on Faehndrich *et al.*'s work by making online cycle detection more complete and efficient [6,7,10,11]. In particular, Heintze and Tardieu [7] describe a field-based analysis, which is capable of analyzing over 1 million lines of C code in a matter of seconds. Field-based analysis does not always meet the needs of the client analysis, particularly since field-based analysis is unsound for C; a field-insensitive version of their algorithm is significantly slower [6].

Rountev *et al.* [12] introduce Offline Variable Substitution (OVS), a linear-time static analysis whose aim is to find and collapse pointer-equivalent variables. Of all the related work, OVS is the most similar to our optimizations and serves as the baseline for our experiments in this paper.

Both pointer and location equivalence have been used in other types of pointer analyses, although they have not been explicitly identified as such; Steensgaard's analysis [14], Das' One-Level Flow [4], and the Shapiro-Horwitz family of analyses [13] all sacrifice precision to gain extra performance by inducing artificial pointer and location equivalences. By contrast, we detect and exploit actual equivalences between variables without losing precision.

Location equivalence has also been used by Liang and Harrold to optimize dataflow analyses [8], but only post-pointer analysis. We give the first method for soundly exploiting location equivalence to optimize the pointer analysis itself.

## 3 Pointer Equivalence

Let $\mathcal{V}$ be the set of all program variables; for $v \in \mathcal{V} : pts(v) \subseteq \mathcal{V}$ is $v$'s points-to set, and $pe(v) \in \mathcal{N}$ is the *pointer equivalence label* of $v$, where $\mathcal{N}$ is the set of natural numbers. Variables $x$ and $y$ are pointer equivalent iff $pts(x) = pts(y)$. Our goal is to assign pointer equivalence labels such that $pe(x) = pe(y)$ implies that $x$ and $y$ are pointer equivalent. Pointer equivalent variables can safely be collapsed together in the constraint graph to reduce both the number of nodes and edges in the graph. The benefits are two-fold: (1) there are fewer points-to sets to maintain; and (2) there are fewer propagations of points-to information along the edges of the constraint graph.

The analysis generates inclusion constraints using a linear pass through the program code; control-flow information is discarded and only variable assignments are considered. Function calls and returns are treated as gotos and are broken down into sets of parameter assignments. Table 1 illustrates the types of constraints and defines their meaning.

**Table 1.** Inclusion Constraint Types

| **Program Code** | **Constraint** | **Meaning** |
|---|---|---|
| $a = \&b$ | $a \supseteq \{b\}$ | $b \in pts(a)$ |
| $a = b$ | $a \supseteq b$ | $pts(a) \supseteq pts(b)$ |
| $a = *b$ | $a \supseteq *b$ | $\forall v \in pts(b) : pts(a) \supseteq pts(v)$ |
| $*a = b$ | $*a \supseteq b$ | $\forall v \in pts(a) : pts(v) \supseteq pts(b)$ |

Our optimizations use these constraints to create an *offline constraint graph*,[1] with VAR nodes to represent each variable, REF nodes to represent each dereferenced variable, and ADR nodes to represent each address-taken variable. A REF node $*a$ stands for the unknown points-to set of variable $a$, while ADR node $\&a$ stands for the address of variable $a$. Edges represent the inclusion relationships: $a \supseteq \{b\}$ yields edge $\&b \rightarrow a$; $a \supseteq b$ yields $b \rightarrow a$; $a \supseteq *b$ yields $*b \rightarrow a$; and $*a \supseteq b$ yields $b \rightarrow *a$.

Before describing the optimizations, we first explain the concepts of *direct* and *indirect* nodes [12]. Direct nodes have all of their points-to relations explicitly represented in the constraint graph: for direct node $x$ and the set of nodes $\mathcal{S} = \{y : y \rightarrow x\}$, $pts(x) = \bigcup_{y \in \mathcal{S}} pts(y)$. Indirect nodes are those that may have points-to relations that are not represented in the constraint graph. All REF nodes are indirect because the unknown variables that they represent may have their own points-to relations. VAR nodes are indirect if they (1) have had their address taken, which means that they can be referenced indirectly via a REF node; (2) are the formal parameter of an indirect function call; or (3) are assigned the return value of an indirect function call. All other VAR nodes are direct.

All indirect nodes are conservatively treated as possible sources of points-to information, and therefore each is given a distinct pointer equivalence label at the beginning of the algorithm. ADR nodes are definite sources of points-to information, and they are also given distinct labels. For convenience, we will use the term 'indirect node' to refer to both ADR nodes and true indirect nodes because they will be treated equivalently by our optimizations.

Figure 1 shows a set of constraints and the corresponding offline constraint graph. In Figure 1 all the REF and ADR nodes are marked indirect, as well as VAR nodes $a$ and $d$, because they have their address taken. Because $a$ and $d$ can

[1] The offline constraint graph is akin to the *subset graph* described by Rountev *et al.* [12].

| | | |
|---|---|---|
| $b \supseteq \{a\}$ | $a \supseteq h$ | $h \supseteq *b$ |
| $b \supseteq \{d\}$ | $c \supseteq b$ | $i \supseteq *e$ |
| $c \supseteq \{a\}$ | $d \supseteq i$ | $k \supseteq *j$ |
| $e \supseteq \{a\}$ | $e \supseteq f$ | |
| $e \supseteq \{d\}$ | $f \supseteq e$ | |
| | $g \supseteq f$ | |

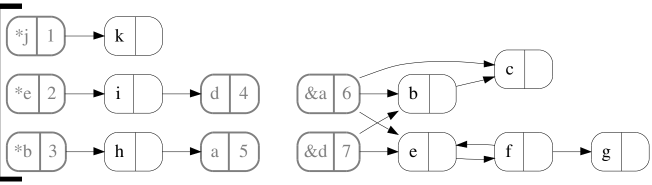


**Fig. 1.** Example offline constraint graph. Indirect nodes are grey and have already been given their pointer equivalence labels. Direct nodes are black and have not been given pointer equivalence labels.

now be accessed indirectly through pointer dereference, we can no longer assume that they only acquire points-to information via nodes $h$ and $i$, respectively.

### 3.1 Hash-Based Value Numbering (HVN)

The goal of HVN is to give each direct node a pointer equivalence label such that two nodes share the same label only if they are pointer equivalent. HVN can also identify non-pointers—variables that are guaranteed to never point to anything. Non-pointers can arise in languages with weak types systems, such as C: the constraint generator can't rely on the variables' type declarations to determine whether a variable is a pointer or not, so it conservatively assumes that everything is a pointer. HVN can eliminate many of these superfluous variables; they are identified by assigning a pointer equivalence label of 0. The algorithm proceeds as follows:

1. Find and collapse strongly-connected components (SCCs) in the offline constraint graph. If any node in the SCC is indirect, the entire SCC is indirect. In Figure 1, $e$ and $f$ are collapsed into a single (direct) node.
2. Proceeding in topological order, for each direct node $x$ let $\mathcal{L}$ be the set of positive incoming pointer equivalence labels, *i.e.* $\mathcal{L} = \{pe(y) : y \rightarrow x \wedge pe(y) \neq 0\}$. There are three cases:
   (a) $\mathcal{L}$ is empty. Then $x$ is a non-pointer and $pe(x) = 0$.
   **Explanation:** in order for $x$ to potentially be a pointer, there must exist a path to $x$ either from an ADR node or some indirect node. If there is no such path, then $x$ must be a non-pointer.
   (b) $\mathcal{L}$ is a singleton, with $p \in \mathcal{L}$. Then $pe(x) = p$.
   **Explanation:** if every points-to set coming in to $x$ is identical, then $x$'s points-to set, being the union of all the incoming points-to sets, must be identical to the incoming sets.
   (c) $\mathcal{L}$ contains multiple labels. The algorithm looks up $\mathcal{L}$ in a hashtable to see if it has encountered the set before. If so, it assigns $pe(x)$ the same label; otherwise it creates a new label, stores it in the hashtable, and assigns it to $pe(x)$.
   **Explanation:** $x$'s points-to set is the union of all the incoming points-to sets; $x$ must be equivalent to any node whose points-to set results from unioning the same incoming points-to sets.

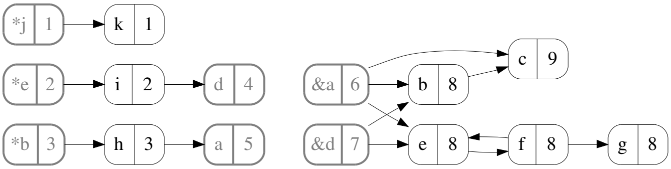


**Fig. 2.** The assignment of pointer equivalence labels after HVN

Following these steps for Figure 1, the final assignment of pointer equivalence labels for the direct nodes is shown in Figure 2. Once we have assigned pointer equivalence labels, we merge nodes with identical labels and eliminate all edges incident to nodes labeled 0.

*Complexity.* The complexity of HVN is linear in the size of the graph. Using Tarjan's algorithm for detecting SCCs [15], step 1 is linear. The algorithm then visits each direct node exactly once and examines its incoming edges. This step is also linear.

*Comparison to OVS.* HVN is similar to Rountev *et al.*'s [12] OVS optimization. The main difference lies in our insight that labeling the condensed offline constraint graph is essentially equivalent to performing value-numbering on a block of straight-line code, and therefore we can adapt the classic compiler optimization of hash-based value numbering for this purpose. The advantage lies in step 2*c*: in this case OVS would give the direct node a new label without checking to see if any other direct nodes have a similar set of incoming labels, potentially missing a pointer equivalence. In the example, OVS would not discover that *b* and *e* are equivalent and would give them different labels.

### 3.2 Extending HVN

HVN does not find all pointer equivalences that can be detected prior to pointer analysis because it does not interpret the *union* and *dereference* operators. Recall that the union operator is implicit in the offline constraint graph: for direct node $x$ with incoming edges from nodes $y$ and $z$, $pts(x) = pts(y) \cup pts(z)$. By interpreting these operators, we can increase the number of pointer equivalences detected, at the cost of additional time and space.

**HR algorithm.** By interpreting the dereference operator, we can relate a VAR node $v$ to its corresponding REF node $*v$. There are two relations of interest:

1. $\forall x, y \in \mathcal{V} : pe(x) = pe(y) \Rightarrow pe(*x) = pe(*y)$.
2. $\forall x \in \mathcal{V} : pe(x) = 0 \Rightarrow pe(*x) = 0$.

The first relation states that if variables $x$ and $y$ are pointer-equivalent, then so are $*x$ and $*y$. If $x$ and $y$ are pointer-equivalent, then by definition $*x$ and

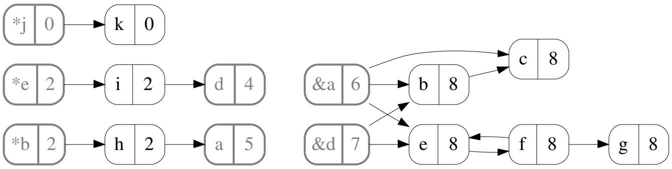


**Fig. 3.** The assignment of pointer equivalence labels after HR and HU

$*y$ will be identical. Whereas HVN would give them unique pointer equivalence labels, we can now assign them the same label. By doing so, we may find additional pointer equivalences that had previously been hidden by the different labels.

The second relation states that if variable $x$ is a non-pointer, then $*x$ is also a non-pointer. It may seem odd to have a constraint that dereferences a non-pointer, but this can happen when code that initializes pointer values is linked but never called, for example with library code. Exposing this relationship can help identify additional non-pointers and pointer equivalences.

Figure 3 provides an example. HVN assigns $b$ and $e$ identical labels; the first relation above tells us we can assign $*b$ and $*e$ identical labels, which exposes the fact that $i$ and $h$ are equivalent to each other, which HVN missed. Also, variable $j$ is not mentioned in the constraints, and therefore the VAR node $j$ isn't shown in the graph, and it is assigned a pointer equivalence label of 0. The second relation above tells us that because $pe(j) = 0$, $pe(*j)$ should also be 0; therefore both $*j$ and $k$ are non-pointers and can be eliminated.

The simplest method for interpreting the dereference operator is to iteratively apply HVN to its own output until it converges to a fixed point. Each iteration collapses equivalent variables and eliminates non-pointers, fulfilling the two relations we describe. This method adds an additional factor of $O(n)$ to the complexity of the algorithm, since in the worst case it eliminates a single variable in each iteration until there is only one variable left. The complexity of HR is therefore $O(n^2)$, but in practice we observe that this method generally exhibits linear behavior.

**HU algorithm.** By interpreting the union operator, we can more precisely track the relations among points-to sets. Figure 3 gives an example in VAR node $c$. Two different pointer equivalence labels reach $c$, one from $\&a$ and one from $b$. HVN therefore gives $c$ a new pointer equivalence label. However, $pts(b) \supseteq pts(\&a)$, so when they are unioned together the result is simply $pts(b)$. By keeping track of this fact, we can assign $c$ the same pointer equivalence label as $b$.

Let $f_n$ be a fresh number unique to $n$; the algorithm will use these fresh values to represent unknown points-to information. The algorithm operates on the condensed offline constraint graph as follows:

1. Initialize points-to sets for each node. $\forall v \in \mathcal{V} : pts(\&v) = \{v\}$; $pts(*v) = \{f_{*v}\}$; if $v$ is direct then $pts(v) = \emptyset$, else $pts(v) = \{f_v\}$.

2. In topological order: for each node $x$, let $\mathcal{S} = \{y : y \to x\} \cup \{x\}$. Then $pts(x) = \bigcup_{y \in \mathcal{S}} pts(y)$.
3. Assign labels s.t. $\forall x, y \in V : pts(x) = pts(y) \Leftrightarrow pe(x) = pe(y)$.

Since this algorithm is effectively computing the transitive closure of the constraint graph, it has a complexity of $O(n^3)$. While this is the same complexity as the pointer analysis itself, HU is significantly faster because, unlike the pointer analysis, we do not add additional edges to the offline constraint graph, making the offline graph much smaller than the graph used by the pointer analysis.

**Putting It Together: HRU.** The HRU algorithm combines the HR and HU algorithms to interpret both the dereference and union operators. HRU modifies HR to iteratively apply the HU algorithm to its own output until it converges to a fixed point. Since the HU algorithm is $O(n^3)$ and HR adds a factor of $O(n)$, HRU has a complexity of $O(n^4)$. As with HR this worst-case complexity is not observed in practice; however it is advisable to first apply HVN to the original constraints, then apply HRU to the resulting set of constraints. HVN significantly decreases the size of the offline constraint graph, which decreases both the time and memory consumption of HRU.

## 4 Location Equivalence

Let $\mathcal{V}$ be the set of all program variables; for $v \in \mathcal{V} : pts(v) \subseteq \mathcal{V}$ is $v$'s points-to set, and $le(v) \in \mathcal{N}$ is the *location equivalence label* of $v$, where $\mathcal{N}$ is the set of natural numbers. Variables $x$ and $y$ are location equivalent iff $\forall z \in \mathcal{V} : x \in pts(z) \Leftrightarrow y \in pts(z)$. Our goal is to assign location equivalence labels such that $le(x) = le(y)$ implies that $x$ and $y$ are location equivalent. Location equivalent variables can safely be collapsed together in all points-to sets, providing two benefits: (1) the points-to sets consume less memory; and (2) since the points-to sets are smaller, points-to information is propagated more efficiently across the edges of the constraint graph.

Without any pointer information it is impossible to compute all location equivalences. However, since points-to sets are never split during the pointer analysis, any variables that are location equivalent at the beginning are guaranteed to be location equivalent at the end. We can therefore safely compute a subset of the equivalences prior to the pointer analysis. We use the same offline constraint graph as we use to find pointer equivalence, but we will be labeling ADR nodes instead of direct nodes. The algorithm assigns each ADR node a label based on its outgoing edges such that two ADR nodes have the same label iff they have the same set of outgoing edges. In other words, ADR nodes $\&a$ and $\&b$ are assigned the same label iff, in the constraints, $\forall z \in \mathcal{V} : z \supseteq \{a\} \Leftrightarrow z \supseteq \{b\}$. In Figure 1, the ADR nodes $\&a$ and $\&d$ would be assigned the same location equivalence label.

While location and pointer equivalences can be computed independently, it is more precise to compute location equivalence *after* we have computed pointer equivalence. We modify the criterion to require that ADR nodes &$a$ and &$b$ are assigned the same label iff $\forall y, z \in V, (y \supseteq \{a\} \wedge z \supseteq \{b\}) \Rightarrow pe(y) = pe(z)$. In other words, we don't require that the two ADR nodes have the same set of outgoing edges, but rather that the nodes incident to the ADR nodes have the same set of pointer equivalence labels.

Once the algorithm has assigned location equivalence labels, it merges all ADR nodes that have identical labels. These merged ADR nodes are each given a fresh name. Points-to set elements will come from this new set of fresh names rather than from the original names of the merged ADR nodes, thereby saving space, since a single fresh name corresponds to multiple ADR nodes. However, we must make a simple change to the subsequent pointer analysis to accommodate this new naming scheme. When adding new edges from indirect constraints, the pointer analysis must translate from the fresh names in the points-to sets to the original names corresponding to the VAR nodes in the constraint graph. To facilitate this translation we create a one-to-many mapping between the fresh names and the original ADR nodes that were merged together. In Figure 1, since ADR nodes &$a$ and &$d$ are given the same location equivalence label, they will be merged together and assigned a fresh name such as &$l$. Any points-to sets that formerly would have contained $a$ and $d$ will instead contain $l$; when adding additional edges from an indirect constraint that references $l$, the pointer analysis will translate $l$ back to $a$ and $d$ to correctly place the edges in the online constraint graph.

*Complexity.* LE is linear in the size of the constraint graph. The algorithm scans through the constraints, and for each constraint $a \supseteq \{b\}$ it inserts $pe(a)$ into ADR node &$b$'s set of pointer equivalence labels. This step is linear in the number of constraints (*i.e.* graph edges). It then visits each ADR node, and it uses a hash table to map from that node's set of pointer equivalence labels to a single location equivalence label. This step is also linear.

# 5 Evaluation

## 5.1 Methodology

Using a suite of six open-source C programs, which range in size from 169K to 2.17M LOC, we compare the analysis times and memory consumption of OVS, HVN, HRU, and HRU+LE (HRU coupled with LE). We then use three different state-of-the-art inclusion-based pointer analyses—Pearce *et al.* [10] (PKH), Heintze and Tardieu [7] (HT), and Hardekopf and Lin [6] (HL)—to compare the optimizations' effects on the pointer analyses' analysis time and memory consumption. These pointer analyses are all field-insensitive and implemented in a common framework, re-using as much code as possible to provide a fair comparison. The source code is available from the authors upon request.

The offline optimizations and the pointer analyses are written in C++ and handle all aspects of the C language except for varargs. We use sparse bitmaps taken from GCC 4.1.1 to represent the constraint graph and points-to sets. The constraint generator is separate from the constraint solvers; we generate constraints from the benchmarks using the CIL C front-end [9], ignoring any assignments involving types too small to hold a pointer. External library calls are summarized using hand-crafted function stubs.

The benchmarks for our experiments are described in Table 2. We run the experiments on an Intel Core Duo 1.83 GHz processor with 2 GB of memory, using the Ubuntu 6.10 Linux distribution. Though the processor is dual-core, the executables themselves are single-threaded. All executables are compiled with GCC 4.1.1 and the '–O3' optimization flag. We repeat each experiment three times and report the smallest time; all the experiments have very low variance in performance. Times include everything from reading the constraint file from disk to computing the final solution.

**Table 2.** Benchmarks: For each benchmark we show the number of lines of code (computed as the number of non-blank, non-comment lines in the source files), a description of the benchmark, and the number of constraints generated by the CIL front-end

| Name | Description | LOC | Constraints |
|---:|:---|---:|---:|
| Emacs-21.4a | text editor | 169K | 83,213 |
| Ghostscript-8.15 | postscript viewer | 242K | 169,312 |
| Gimp-2.2.8 | image manipulation | 554K | 411,783 |
| Insight-6.5 | graphical debugger | 603K | 243,404 |
| Wine-0.9.21 | windows emulator | 1,338K | 713,065 |
| Linux-2.4.26 | linux kernel | 2,172K | 574,788 |

### 5.2 Cost of Optimizations

Tables 3 and 4 show the analysis time and memory consumption, respectively, of the offline optimizations on the six benchmarks. OVS and HVN have roughly the same times, with HVN using 1.17× more memory than OVS. On average, HRU and HRU+LE are 3.1× slower and 3.4× slower than OVS, respectively. Both HRU and HRU+LE have the same memory consumption as HVN. As stated earlier, these algorithms are run on the output of HVN in order to improve analysis time and conserve memory; their times are the sum of their running time and the HVN running time, while their memory consumption is the maximum of their memory usage and the HVN memory usage. In all cases, the HVN memory usage is greater.

Figure 4 shows the effect of each optimization on the number of constraints for each benchmark. On average OVS reduces the number of constraints by 63.4%, HVN by 69.4%, HRU by 77.4%, and HRU+LE by 79.9%. HRU+LE, our most aggressive optimization, takes 3.4× longer than OVS, while it only reduces the number of constraints by an additional 16.5%. However, inclusion-based analysis is $O(n^3)$ time and $O(n^2)$ space, so even a relatively small reduction in the input size can have a significant effect, as we'll see in the next section.

**Table 3.** Offline analysis times (sec)

| | Emacs | Ghostscript | Gimp | Insight | Wine | Linux |
|---|---|---|---|---|---|---|
| **OVS** | 0.29 | 0.60 | 1.74 | 0.96 | 3.57 | 2.34 |
| **HVN** | 0.29 | 0.61 | 1.66 | 0.95 | 3.39 | 2.36 |
| **HRU** | 0.49 | 2.29 | 4.31 | 4.28 | 9.46 | 7.70 |
| **HRU+LE** | 0.53 | 2.54 | 4.75 | 4.64 | 10.41 | 8.47 |

**Table 4.** Offline analysis memory (MB)

| | Emacs | Ghostscript | Gimp | Insight | Wine | Linux |
|---|---|---|---|---|---|---|
| **OVS** | 13.1 | 28.1 | 61.1 | 39.1 | 110.4 | 96.2 |
| **HVN** | 14.8 | 32.5 | 71.5 | 44.7 | 134.8 | 114.8 |
| **HRU** | 14.8 | 32.5 | 71.5 | 44.7 | 134.8 | 114.8 |
| **HRU+LE** | 14.8 | 32.5 | 71.5 | 44.7 | 134.8 | 114.8 |

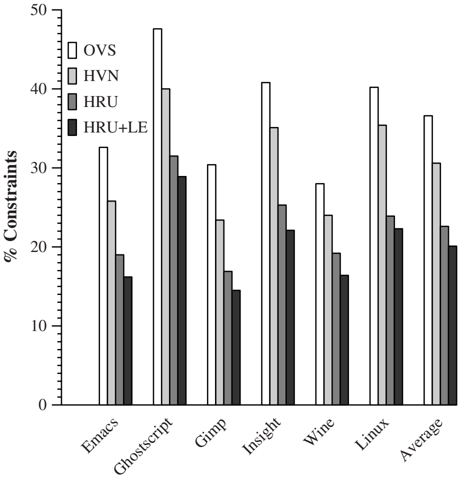


**Fig. 4.** Percent of the original number of constraints that is generated by each optimization

### 5.3 Benefit of Optimizations

Tables 5–10 give the analysis times and memory consumption for three pointer analyses—PKH, HT, and HL—as run on the results of each offline optimization; OOM indicates the analysis ran out of memory. The data is summarized in Figure 5, which gives the average performance and memory improvement for the three pointer analyses for each offline algorithm as compared to OVS. The offline analysis times are added to the pointer analysis times to make the overall analysis time comparison.

**Table 5.** Online analysis times for the PKH algorithm (sec)

| | **Emacs** | **Ghostscript** | **Gimp** | **Insight** | **Wine** | **Linux** |
|---|---|---|---|---|---|---|
| **OVS** | 1.99 | 19.15 | 99.22 | 121.53 | 1,980.04 | 1,202.78 |
| **HVN** | 1.60 | 17.08 | 87.03 | 111.81 | 1,793.17 | 1,126.90 |
| **HRU** | 0.74 | 13.31 | 38.54 | 57.94 | 1,072.18 | 598.01 |
| **HRU+LE** | 0.74 | 9.50 | 21.03 | 33.72 | 731.49 | 410.23 |

**Table 6.** Online analysis memory for the PKH algorithm (MB)

| | **Emacs** | **Ghostscript** | **Gimp** | **Insight** | **Wine** | **Linux** |
|---|---|---|---|---|---|---|
| **OVS** | 23.1 | 102.7 | 418.1 | 251.4 | 1,779.7 | 1,016.5 |
| **HVN** | 17.7 | 83.9 | 269.5 | 194.8 | 1,448.5 | 840.8 |
| **HRU** | 12.8 | 68.0 | 171.6 | 165.4 | 1,193.7 | 590.4 |
| **HRU+LE** | 6.9 | 23.8 | 56.1 | 58.6 | 295.9 | 212.4 |

**Table 7.** Online analysis times for the HT algorithm (sec)

| | **Emacs** | **Ghostscript** | **Gimp** | **Insight** | **Wine** | **Linux** |
|---|---|---|---|---|---|---|
| **OVS** | 1.63 | 13.58 | 64.45 | 46.32 | **OOM** | 410.52 |
| **HVN** | 1.84 | 12.84 | 59.68 | 42.70 | **OOM** | 393.00 |
| **HRU** | 0.70 | 9.95 | 37.27 | 37.03 | 1,087.84 | 464.51 |
| **HRU+LE** | 0.54 | 8.82 | 18.71 | 23.35 | 656.65 | 332.36 |

**Table 8.** Online analysis memory for the HT algorithm (MB)

| | **Emacs** | **Ghostscript** | **Gimp** | **Insight** | **Wine** | **Linux** |
|---|---|---|---|---|---|---|
| **OVS** | 22.5 | 97.2 | 359.7 | 266.9 | **OOM** | 1,006.8 |
| **HVN** | 17.7 | 85.0 | 279.0 | 231.5 | **OOM** | 901.3 |
| **HRU** | 10.8 | 70.3 | 205.3 | 156.7 | 1,533.0 | 700.7 |
| **HRU+LE** | 6.4 | 34.9 | 86.0 | 69.4 | 820.9 | 372.2 |

**Table 9.** Online analysis times for the HL algorithm (sec)

| | **Emacs** | **Ghostscript** | **Gimp** | **Insight** | **Wine** | **Linux** |
|---|---|---|---|---|---|---|
| **OVS** | 1.07 | 9.15 | 17.55 | 20.45 | 534.81 | 103.37 |
| **HVN** | 0.68 | 8.14 | 13.69 | 17.23 | 525.31 | 91.76 |
| **HRU** | 0.32 | 7.25 | 10.04 | 12.70 | 457.49 | 75.21 |
| **HRU+LE** | 0.51 | 6.67 | 8.39 | 13.71 | 345.56 | 79.99 |

*Analysis Time.* For all three pointer analyses, HVN only moderately improves analysis time over OVS, by 1.03–1.18×. HRU has a greater effect despite its much higher offline analysis times; it improves analysis time by 1.28–1.88×. HRU+LE has the greatest effect; it improves analysis time by 1.28–2.68×. An important factor in the analysis time of these algorithms is the number of times

**Table 10.** Online analysis memory for the HL algorithm (MB)

| | **Emacs** | **Ghostscript** | **Gimp** | **Insight** | **Wine** | **Linux** |
|---|---|---|---|---|---|---|
| **OVS** | 21.0 | 93.9 | 415.4 | 239.7 | 1,746.3 | 987.8 |
| **HVN** | 13.9 | 73.5 | 263.9 | 183.7 | 1,463.5 | 807.9 |
| **HRU** | 9.2 | 63.3 | 170.7 | 121.9 | 1,185.3 | 566.6 |
| **HRU+LE** | 4.5 | 22.2 | 33.4 | 27.6 | 333.1 | 162.6 |

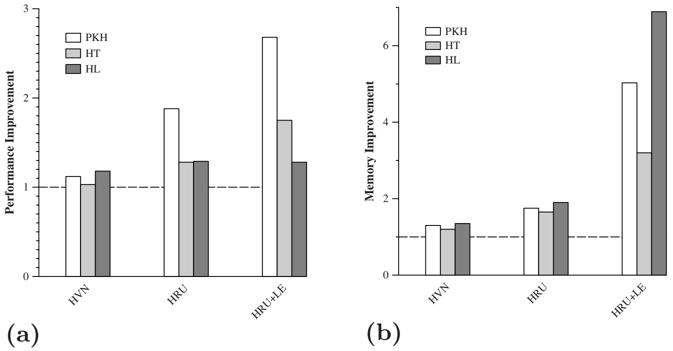


**Fig. 5.** **(a)** Average performance improvement over OVS; **(b)** Average memory improvement over OVS. For each graph, and for each offline optimization $X \in$ {HVN, HRU, HRU+LE}, we compute $\frac{OVS_{time/memory}}{X_{time/memory}}$.

they propagate points-to information across constraint edges. PKH is the least efficient of the algorithms in this respect, propagating much more information than the other two; hence it benefits more from the offline optimizations. HL propagates the least amount of information and therefore benefits the least.

*Memory.* For all three pointer analyses HVN only moderately improves memory consumption over OVS, by 1.2–1.35×. All the algorithms benefit significantly from HRU, using 1.65–1.90× less memory than for OVS. HRU's greater reduction in constraints makes for a smaller constraint graph and fewer points-to sets. HRU+LE has an even greater effect: HT uses 3.2× less memory, PKH uses 5× less memory, and HL uses almost 7× less memory. HRU+LE doesn't further reduce the constraint graph or the number of points-to sets, but on average it cuts the average points-to set size in half.

*Room for Improvement.* Despite aggressive offline optimization in the form of HRU plus the efforts of online cycle detection, there are still a significant number of pointer equivalences that we do not detect in the final constraint graph. The number of actual pointer equivalence classes is much smaller than the number of detected equivalence classes, by almost 4× on average. In other words, we could conceivably shrink the online constraint graph by almost 4× if we could do a better job of finding pointer equivalences. This is an interesting area for future work. On the other hand, we do detect a significant fraction of the actual location equivalences—we detect 90% of the actual location equivalences in the five largest benchmarks, though for the smallest (Emacs) we only detect 41%. Thus there is not much room to improve on the LE optimization.

**Bitmaps vs. BDDs.** The data structure used to represent points-to sets for the pointer analysis can have a great effect on the analysis time and memory consumption of the analysis. Hardekopf and Lin [6] compare the use of sparse bitmaps versus BDDs to represent points-to sets and find that on average the BDD implementation is 2× slower but uses 5.5× less memory than the bitmap implementation. To make a similar comparison testing the effects of our optimizations, we implement two versions of each pointer analysis: one using sparse bitmaps to represent points-to sets, the other using BDDs for the same purpose. Unlike BDD-based pointer analyses [2,16] which store the entire points-to solution in a single BDD, we give each variable its own BDD to store its individual points-to set. For example, if $v \rightarrow \{w, x\}$ and $y \rightarrow \{x, z\}$, the BDD-based analyses would have a single BDD that represents the set of tuples $\{(v, w), (v, x), (y, x), (y, z)\}$. Instead, we give $v$ a BDD that represents the set $\{w, x\}$ and we give $y$ a BDD that represents the set $\{w, z\}$. The two BDD representations take equivalent memory, but our representation is a simple modification that requires minimal changes to the existing code.

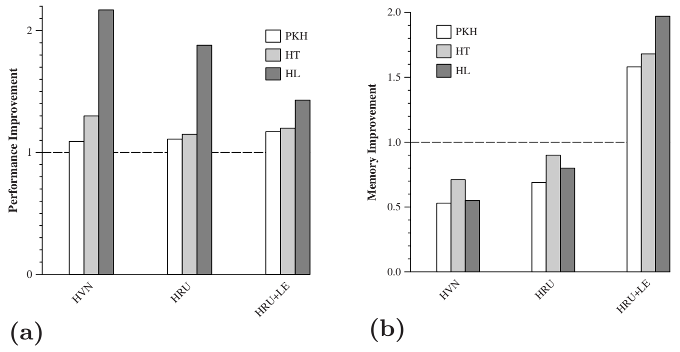


**Fig. 6. (a)** Average performance improvement over BDDs; **(b)** Average memory improvement over BDDs. Let $BDD$ be the BDD implementation and $BIT$ be the bitmap implementation; for each graph we compute $\frac{BDD_{time/memory}}{BIT_{time/memory}}$.

The results of our comparison are shown in Figure 6. We find that for HVN and HRU, the bitmap implementations on average are 1.4–1.5× faster than the BDD implementations but use 3.5–4.4× more memory. However, for HRU+LE the bitmap implementations are on average 1.3× faster and use 1.7× *less* memory than the BDD implementations, because the LE optimization significantly shrinks the points-to sets of the variables.

## 6 Conclusion

In this paper we have shown that it is possible to reduce both the memory consumption and analysis time of inclusion-based pointer analysis without affecting precision. We have empirically shown that for three well-known inclusion-based analyses with highly tuned implementations, our offline optimizations improve average analysis time by 1.3–2.7× and reduce average memory consumption by 3.2–6.9×. For the fastest known inclusion-based analysis [6], the optimizations improve analysis time by 1.3× and reduce memory consumption by 6.9×. We have also found the somewhat surprising result that with our optimizations a sparse bitmap representation of points-to sets is both faster and requires less memory than a BDD representation.

In addition, we have provided a roadmap for further investigations into the optimization of inclusion-based analysis. Our optimization that exploits location equivalence comes close to the limit of what can be accomplished, but our other optimizations identify only a small fraction of the pointer equivalences. Thus, the exploration of new methods for finding and exploiting pointer equivalences should be a fruitful area for future work.

*Acknowledgments.* We thank Brandon Streiff and Luke Robison for their help in conducting experiments and Dan Berlin for his help with the GCC compiler internals. Kathryn McKinley, Ben Wiedermann, and Adam Brown provided valuable comments on earlier drafts. This work was supported by NSF grant ACI-0313263 and a grant from the Intel Research Council.

## References

1. Andersen, L.O.: Program Analysis and Specialization for the C Programming anguage. PhD thesis, DIKU, University of Copenhagen (May 1994)
2. Berndl, M., Lhotak, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using BDDs. In: Programming Language Design and Implementation (PLDI), pp. 103–114 (2003)
3. Briggs, P., Cooper, K.D., Taylor Simpson, L.: Value numbering. Software Practice and Experience 27(6), 701–724 (1997)
4. Das, M.: Unification-based pointer analysis with directional assignments. In: Programming Language Design and Implementation (PLDI), pp. 35–46 (2000)
5. Faehndrich, M., Foster, J.S., Su, Z., Aiken, A.: Partial online cycle elimination in inclusion constraint graphs. In: Programming Language Design and Implementation (PLDI), pp. 85–96 (1998)
6. Hardekopf, B., Lin, C.: The Ant and the Grasshopper: Fast and accurate pointer analysis for millions of lines of code. In: Programming Language Design and Implementation (PLDI) (2007)
7. Heintze, N., Tardieu, O.: Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In: Programming Language Design and Implementation (PLDI), pp. 24–34 (2001)
8. Liang, D., Harrold, M.J.: Equivalence analysis and its application in improving the efficiency of program slicing. ACM Trans. Softw. Eng. Methodol. 11(3), 347–383 (2002)

9. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Computational Complexity, pp. 213–228 (2002)
10. Pearce, D., Kelly, P., Hankin, C.: Efficient field-sensitive pointer analysis for C. In: ACM workshop on Program Analysis for Software Tools and Engineering (PASTE), pp. 37–42. ACM Press, New York (2004)
11. Pearce, D.J., Kelly, P.H.J., Hankin, C.: Online cycle detection and difference propagation for pointer analysis. In: 3rd International IEEE Workshop on Source Code Analysis and Manipulation (SCAM), pp. 3–12. IEEE Computer Society Press, Los Alamitos (2003)
12. Rountev, A., Chandra, S.: Off-line variable substitution for scaling points-to analysis. In: Programming Language Design and Implementation (PLDI), pp. 47–56 (2000)
13. Shapiro, M., Horwitz, S.: Fast and accurate flow-insensitive points-to analysis. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 1–14. ACM Press, New York (1997)
14. Steensgaard, B.: Points-to analysis in almost linear time. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 32–41. ACM Press, New York (1996)
15. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comput. 1(2), 146–160 (June 1972)
16. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis. In: Programming Language Design and Implementation (PLDI), pp. 131–144 (2004)

# Hierarchical Pointer Analysis for Distributed Programs

Amir Kamil and Katherine Yelick

Computer Science Division, University of California, Berkeley
{kamil,yelick}@cs.berkeley.edu

**Abstract.** We present a new pointer analysis for use in shared memory programs running on hierarchical parallel machines. The analysis is motivated by the partitioned global address space languages, in which programmers have control over data layout and threads and can directly read and write to memory associated with other threads. Titanium, UPC, Co-Array Fortran, X10, Chapel, and Fortress are all examples of such languages. The novelty of our analysis comes from the hierarchical machine model used, which captures the increasingly hierarchical nature of modern parallel machines. For example, the analysis can distinguish between pointers that can reference values within a thread, within a shared memory multiprocessor, or within a network of processors. The analysis is presented with a formal type system and operational semantics, articulating the various ways in which pointers can be used within a hierarchical machine model. The hierarchical analysis has several applications, including race detection, sequential consistency enforcement, and software caching. We present results of an implementation of the analysis, applying it to data race detection, and show that the hierarchical analysis is very effective at reducing the number of false races detected.

## 1 Introduction

The introduction of multi-core processors marks a dramatic shift in software development: parallel software will be required for laptops, desktops, gaming consoles, and graphics processors. These chips are building blocks in larger shared and distributed memory parallel systems, resulting in machines that are increasingly hierarchical and use a combination of cache-coherent shared memory, partitioned memory with (remote) direct memory access (DMA or RDMA), and message passing. The partitioned global address space (PGAS) model is a natural fit for programming these machines, and languages that use it include Unified Parallel C (UPC) [7,26], Co-Array Fortran (CAF) [25], Titanium [28,12] (based on Java [10]), Chapel [8], X10 [24], and Fortress [1]. In all of these languages, pointers to shared state is permitted, and a fundamental question is whether a given pointer can be proven to access data in only a limited part of the machine hierarchy. Some applications of this are: 1) a pointer that accesses data that is private to a single thread cannot be involved in a data race; 2) a pointer that accesses data within a chip multiprocessor may require memory fences to ensure ordering, but those fences only need to make data visible within the chip level; 3) pointer limits may inform a software caching system that coherence protocols may be restricted to a subset of processors; 4) a pointer with a limited domain may use fewer bits in its representation, since only a fraction of the total address space is accessible.

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 281–297, 2007.

In this paper we introduce a pointer analysis that is designed for a hierarchical setting. Our analysis allows for an arbitrarily deep hierarchy, such as the abstract machine model in Fortress, although in this paper we apply it to the three-level model of Titanium. In Titanium, a pointer may refer to data only within a single thread, or to data associated with any threads within a SMP node, or to any thread in the machine.

We develop a model language, *Ti*, for presenting our analysis and give both a type system and operational semantics for the language. *Ti* has the essential features of any global address space language: the ability to create references to data, share data with other machines in the system through references, and dereference them for either read or write access. *Ti* also has a hierarchical machine model, which is general enough to cover all of the existing PGAS languages. We implement our analysis in the context of the full Titanium language and then apply the analysis, in conjunction with an existing concurrency analysis [15], towards race detection, and show that it greatly reduces the number of false races detected on five application benchmarks. In previous work we demonstrated some of the other applications of pointer analysis in Titanium [14], but without the generality of the hierarchical analysis presented here.

## 2 Background

In this section, we describe some machines and languages that use a hierarchical memory model and discuss the aspects of Titanium that are relevant to the pointer analysis.

### 2.1 Hierarchical Memory

Parallel machines are often built with hierarchical memory systems, with local caches or explicitly managed local stores associated with each process. For example, partitioned global address space (PGAS) languages may run on shared memory, distributed memory machines or hybrids, with the language runtime providing the illusion of shared memory through the use of wide pointers (that store both a processor node number and an address), distributed arrays, and implicit communication to access such data. Hierarchies also exist within processors in the form of caches and local stores. For example, the Cell game processor has a local store associated with each of the *SPE* processors, which can be accessed by other SPEs through memory move (DMA) operations. Additional levels of partitioning are also possible, such as partitioning memory in a computational grid into clusters, each of which is partitioned into nodes, as in Figure 1.

Most PGAS languages use a two level abstraction of memory, where data is either local to a thread or shared by all, although Titanium uses three levels and Fortress has an arbitrary number. In many PGAS languages, pointers are restricted in what they can reference. In Figure 1, pointers A, B, and C are examples of pointers that can only refer to thread-local, node-local, and cluster-local locations, respectively, while D can point anywhere in the grid. The *width* of a pointer specifies what locations it can reference, with a higher width allowing further locations, as shown by the edge labels in Figure 1. Wider pointers consume more space and are more expensive to manipulate and access. For example, thread-local and node-local pointers could be represented simply by an address, while a cluster-local pointer contains an address and a node number. Wider pointers also have added costs to dereference, even if they happen to refer

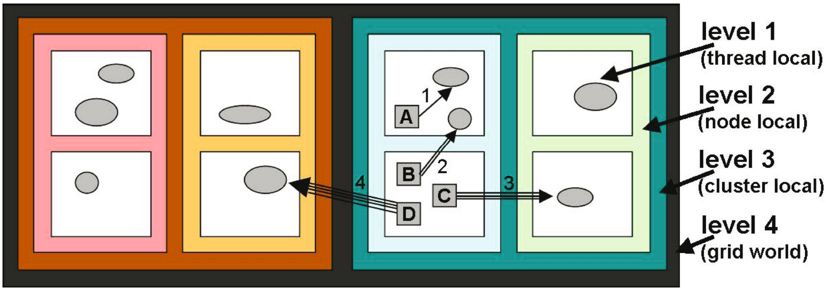


**Fig. 1.** A possible machine hierarchy with four levels. The width of arrows and their labels indicate the hierarchy distance between the endpoints.

to nearby data; the pointer must be checked to see whether it is local, and coherence traffic or fences may be required to ensure the data is consistent with that viewed by other threads. The trend in hardware is towards more levels of hierarchy, and towards high costs between levels. Thus, software that can take advantage of the hierarchy is increasingly important.

## 2.2 Titanium

The Titanium programming language [28] is a high performance dialect of Java designed for distributed machines. It is a *single program, multiple data* (SPMD) language, so all threads execute the same code image. In addition, Titanium has a global address space abstraction, so that any thread can directly access memory on another thread. At runtime, two threads may share the same physical address space, in which case such an access is done directly, or they may be in distinct address spaces, in which case the global access must be translated into communication through the GASNet communication layer [6].

In addition to dereferencing, communication between threads can be done through the one-to-all *broadcast* and the all-to-all *exchange* operations. Program variables, including static variables, are not shared between threads, so they cannot be used for communication.

Since threads can share a physical address space, they are arranged in the following three-level hierarchy:

- **Level 1:** an individual thread
- **Level 2:** threads within the same physical address space
- **Level 3:** all threads

In the Titanium type system, variables are implicitly *global*, meaning that they can point to a location on any thread (level 3). A variable can be restricted to only point within a physical address space (level 2) by qualifying it with the `local` keyword. Downcasts between global and local are allowed and only succeed if the actual location referenced is within the same physical address space as the executing thread. Our analysis takes advantage of existing such casts in a program in determining what variables must reference data in the same address space.

The Titanium type system does not separate levels 1 and 2 of the hierarchy. The distinction between 1 and 2 is important for many applications, such as race detection [21], data sharing analysis [17], and sequential consistency enforcement [14], since references to level 1 values on different threads cannot be to the same location. Other applications such as data locality inference [16] can benefit from the distinction between levels 2 and 3. Though we could perform a two-level analysis twice to obtain a three-level analysis, we show in §4.5 that the three-level analysis we have implemented is much more efficient.

## 3 Analysis Background

We define a machine[1] hierarchy and a simple language as the basis of our analysis. This allows the analysis to be applied to languages besides Titanium, and it avoids language constructs that are not crucial to the analysis. While the language we use is SPMD, the analysis can easily be extended to other models of parallelism, though we do not do so here.

### 3.1 Machine Structure

Consider a set of machines arranged in an arbitrary hierarchy, such as that of Figure 1. A *machine* corresponds to a single execution stream within a parallel program. Each machine has a corresponding *machine number*. The *depth* of the hierarchy is the number of levels it contains. The *distance* between machines is equal to the level of the hierarchy containing their least common ancestor. A pointer on a machine $m$ has a corresponding *width*, and it can only refer to locations on machines whose distance from $m$ is no more than the pointer's width

### 3.2 Language

Our analysis is formalized using a simple language, called *Ti*, that illustrates the key features of the analysis. *Ti* is a generalization of the language used by Liblit and Aiken in their work on locality inference [16]. Like Titanium, *Ti* uses a SPMD model of parallelism, so that all machines execute the same program text. The height of the machine hierarchy is known statically, and we will refer to it as $h$ from here on. References thus can have any width in the range $[1, h]$.

The syntax of *Ti* is summarized in Figure 2. Types can be integers or reference types. The latter are parameterized by a width $n$, in the range $[1, h]$. Expressions in *Ti* consist of the following

- integer literals $(n)$
- variables $(x)$. We assume a fixed set of variables of predefined type. We also assume that variables are machine-private.

[1] Throughout this paper, we will use *machine* interchangeably with *thread*.

$$
\begin{aligned}
n \; &::= \text{integer literals}\\
x \; &::= \text{variables}\\
\tau \; &::= \mathit{int} \mid \mathtt{ref}_n\, \tau \qquad \text{(types)}\\
e \; &::= n \mid x \mid \mathtt{new}_l\, \tau \mid \; * e \mid \mathtt{convert}(e, n)\\
&\quad \mid \mathtt{transmit}\, e_1\, \mathtt{from}\, e_2 \mid e_1 ; e_2\\
&\quad \mid x := e \mid e_1 \leftarrow e_2\\
&\qquad\qquad\qquad \text{(expressions)}
\end{aligned}
$$

**Fig. 2.** The syntax of the *Ti* language

$$
\begin{aligned}
\mathit{expand}(\mathtt{ref}_m \tau, n) \; &\equiv \mathtt{ref}_{max(m,n)}\, \tau\\
\mathit{expand}(\tau, n) \; &\equiv \tau \quad \text{otherwise}\\[1em]
\mathit{robust}(\mathtt{ref}_m \tau, n) \; &\equiv \mathit{false} \quad \text{if } m < n\\
\mathit{robust}(\tau, n) \; &\equiv \mathit{true} \quad \text{otherwise}
\end{aligned}
$$

**Fig. 3.** Type manipulating functions

$$\frac{}{\Gamma \vdash n : \mathtt{int}} \qquad \frac{}{\Gamma \vdash \mathtt{new}_l\, \tau : \mathtt{ref}_1\, \tau}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e : \mathtt{ref}_n\, \tau}{\Gamma \vdash \; * e : \mathit{expand}(\tau, n)}$$

$$\frac{\Gamma \vdash e : \mathtt{ref}_n\, \tau}{\Gamma \vdash \mathtt{convert}(e, m) : \mathtt{ref}_m\, \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash \mathtt{transmit}\, e_1\, \mathtt{from}\, e_2 : \mathit{expand}(\tau, h)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 ; e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma(x) = \tau}{\Gamma \vdash x := e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{ref}_n\, \tau \quad \Gamma \vdash e_2 : \tau \quad \mathit{robust}(\tau, n)}{\Gamma \vdash e_1 \leftarrow e_2 : \tau}$$

$$\frac{\Gamma \vdash e : \mathtt{ref}_n\, \tau \quad n < m}{\Gamma \vdash e : \mathtt{ref}_m\, \tau}$$

**Fig. 4.** Type checking rules

- reference allocations ($\mathtt{new}_l\ \tau$). The expression $\mathtt{new}_l\ \tau$ allocates a memory cell of type $\tau$ and returns a reference to the cell. In order to facilitate the pointer analysis in §4, each allocation site is given a unique label $l$.
- dereferencing ($*e$)
- type conversions ($\mathtt{convert}(e, n)$), which widen or narrow the width of an expression, converting its type from $\mathtt{ref}_m\ \tau$ to $\mathtt{ref}_n\ \tau$.
- communication ($\mathtt{transmit}\, e_1\ \mathtt{from}\ e_2$). In $\mathtt{transmit}\, e_1\ \mathtt{from}\ e_2$, machine $e_2$ evaluates the expression $e_1$ and sends the result to the other machines.
- sequencing ($e_1; e_2$)
- assignment to variables ($x := e$)
- assignment through references ($e_1 \leftarrow e_2$). In $e_1 \leftarrow e_2$, $e_2$ is written into the location referred to by $e_1$.

For simplicity, *Ti* does not have conditional statements. Since the analysis is flow-insensitive, conditionals are not essential to it.

The type checking rules for *Ti* are summarized in Figure 4. The rules for integer literals, variables, sequencing, and variable assignments are straightforward.

The allocation expression $\mathtt{new}_l\ \tau$ produces a reference type $\mathtt{ref}_1\ \tau$ of width 1, since the allocated memory is guaranteed to be on the machine that is performing the allocation. Pointer dereferencing is more problematic, however. Consider the situation in Figure 5, where $x$ on machine 0 refers to a location on machine 0 that refers to a location on machine 1. This implies that $x$ has type $\mathtt{ref}_1\ \mathtt{ref}_2\ \tau$. The result of $*x$ should

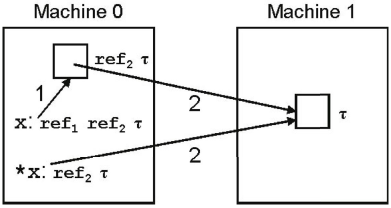


**Fig. 5.** Dereferences may require width expansion. The arrow labels correspond to pointer widths.

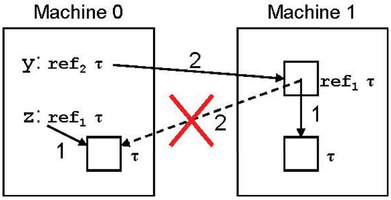


**Fig. 6.** The assignment $y \leftarrow z$ is forbidden, since the location referred to by $y$ can only hold pointers of width 1 but requires a pointer of width 2 to refer to $z$

be a reference to the location on machine 1, so it must have type $\texttt{ref}_2\ \tau$. In general, a dereference of a value of type $\texttt{ref}_a\ \texttt{ref}_b\ \tau$ produces a value of type $\texttt{ref}_{max(a,b)}\ \tau$.

The `convert` expression allows the top-level width of an expression to be up or downcast. Upcasts are rarely used due to the subtyping rule below. A programmer can use downcasts to inform the compiler that the reference is to data residing on a machine closer than the original width, and usually does so only after a dynamic check that this is the case. The resulting type is the same as the input expression, but with the provided top-level width.

In the `transmit` expression, if the value to be communicated is an integer, then the resulting type is still an integer. If the value is a reference, however, the result must be promoted to the maximum width $h$, since the relationship between source and destination is not statically known.

The typing rule for the assignment through reference expression is also nontrivial. Consider the case where $y$ has type $\texttt{ref}_2\ \texttt{ref}_1\ \tau$, as in Figure 6. Should it be possible to assign to $y$ with a value of type $\texttt{ref}_1\ \tau$? Such a value must be on machine 0, but the location referred to by $x$ is on machine 1. Since that location holds a value of type $\texttt{ref}_1\ \tau$, it must refer to a location on machine 1. Thus, the assignment should be forbidden. In general, an assignment to a reference of type $\texttt{ref}_a\ \texttt{ref}_b\ \tau$ should only be allowed if $a \leq b$.

There is also a subtyping rule that allows for implicit widening of a reference. Subsumption is only allowed for the top-level width of a reference.

As in the approach of Liblit and Aiken, [16], we define an *expand* function and a *robust* predicate to facilitate type checking. The *expand* function widens a type when necessary, and the *robust* predicate determines when it is legal to assign to a reference. These functions are shown in Figure 3.

### 3.3 Concrete Operational Semantics

In this section we present the sequential operational semantics of *Ti* . We ignore concurrency in defining the semantics, since it is not essential to our flow-insensitive analysis.

We use the following semantic domains and naming conventions for their elements:

| | |
|---|---|
| $M$ | (the set of machines) |
| $H = \{1, ..., h\}$ | (the set of possible widths) |
| $A$ | (the set of local addresses) |
| $Id$ | (the set of identifiers) |
| $N$ | (the set of integer literals) |
| $Var = M \times Id$ | (the set of variables) |
| $L$ | (the set of allocation site labels) |
| $T$ | (the set of all types) |
| $G = L \times M \times A$ | (the set of global addresses) |
| $V = N \cup G$ | (the set of values) |
| $Store = (G \cup Var) \rightarrow V$ | (the contents of memory) |
| $Exp$ | (the set of all expressions) |

| | |
|---|---|
| $m \in M$ | (a machine) |
| $v \in V$ | (a value) |
| $\sigma \in Store$ | (a memory state) |
| $a \in A$ | (a local address) |
| $l \in L$ | (a label) |
| $g = (l, m, a) \in G$ | (a global address) |
| $e \in Exp$ | (an expression) |

Judgments in our operational semantics have the form $\langle e, m, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$, which means that expression $e$ executed on machine $m$ in a global state $\sigma$ evaluates to the value $v$ and results in the new state $\sigma'$. We use the notation $\sigma[g := v]$ to denote the function $\lambda x.\ \texttt{if}\ x = g\ \texttt{then}\ v\ \texttt{else}\ \sigma(x)$.

The rules for integer and variable expressions are trivial.

$$\overline{\langle n, m, \sigma\rangle \Downarrow \langle n, \sigma\rangle} \qquad \overline{\langle x, m, \sigma\rangle \Downarrow \langle \sigma(x), \sigma\rangle}$$

For allocations, we introduce a special $null$ value to represent uninitialized pointers. The result of an allocation is an address on the local machine that is guaranteed to not already be in use.

$$\overline{\langle new_l\ \tau, m, \sigma\rangle \Downarrow \langle (l, m, a), \sigma[(l, m, a) := null]\rangle}\ (a \text{ is fresh on } m)$$

The rule for dereferencing is simple, except that it is illegal to dereference a $null$ pointer.

$$\frac{\langle e, m, \sigma\rangle \Downarrow \langle g, \sigma'\rangle \quad g \neq null}{\langle *e, m, \sigma\rangle \Downarrow \langle \sigma'(g), \sigma'\rangle}$$

The rule for variable assignment is also simple.

$$\frac{\langle e, m, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle x := e, m, \sigma\rangle \Downarrow \langle v, \sigma'[x := v]\rangle}$$

The rule for assignment through a reference is the combination of a dereference and a normal assignment.

$$\frac{\langle e_1, m, \sigma\rangle \Downarrow \langle g, \sigma_1\rangle \quad \langle e_2, m, \sigma_1\rangle \Downarrow \langle v, \sigma_2\rangle \quad g \neq null}{\langle e_1 \leftarrow e_2, m, \sigma\rangle \Downarrow \langle v, \sigma_2[g := v]\rangle}$$

The rule for sequencing is as expected.

$$\frac{\langle e_1, m, \sigma\rangle \Downarrow \langle v_1, \sigma_1\rangle \quad \langle e_2, m, \sigma_1\rangle \Downarrow \langle v_2, \sigma_2\rangle}{\langle e_1; e_2, m, \sigma\rangle \Downarrow \langle v_2, \sigma_2\rangle}$$

The type conversion expression makes use of the $hier$ function, which returns the hierarchical distance between two machines. The conversion is only allowed if that distance is no more than the target type.

$$\frac{\langle e, m, \sigma\rangle \Downarrow \langle g = (l, m', a), \sigma'\rangle \quad hier(m, m') \leq n}{\langle \texttt{convert}(e, n), m, \sigma\rangle \Downarrow \langle g, \sigma'\rangle}$$

In the `transmit` operation, the expression is evaluated on the given machine.

$$\frac{\langle e_2, m, \sigma\rangle \Downarrow \langle n, \sigma_2\rangle \quad n \in M \quad \langle e_1, n, \sigma_2\rangle \Downarrow \langle v, \sigma_1\rangle}{\langle \texttt{transmit}\, e_1\, \texttt{from}\, e_2, m, \sigma\rangle \Downarrow \langle v, \sigma_1\rangle}$$

## 4 Abstract Interpretation

We now present a pointer analysis for the *Ti* language. So that we can ignore any issues of concurrency and also for efficiency, our analysis is flow-insensitive. We only define the analysis on the single machine $m$ – since *Ti* is SPMD, the results are the same for all machines.

### 4.1 Concrete Domain

Since our analysis is flow-insensitive, we need not determine the concrete state at each point in a program. Instead, we define the concrete state over the whole program. Since we are doing pointer analysis, we are only interested in reference values, and since a location can contain different values over the lifetime of the program, we must compute the set of all possible values for each memory location and variable on machine $m$. The concrete state thus maps each memory location and variable to a set of memory locations, and it is a member of the domain $CS = (G + Id) \rightarrow \mathcal{P}(G)$.

### 4.2 Abstract Domain

For our abstract semantics, we define an *abstract location* to correspond to the abstraction of a concrete memory location. Abstract locations are defined relative to a particular machine $m$. An abstract location relative to machine $m$ is a member of the domain $A_m = L \times H$ – it is identified by both an allocation site and a hierarchy width. An element $a_1$ of $A_m$ is subsumed by another element $a_2$ if $a_1$ and $a_2$ have the same allocation site, and $a_2$ has a higher width than $a_1$. The elements of $A_m$ are thus ordered by the following relation:

$$(l, n_1) \sqsubseteq (l, n_2) \Longleftrightarrow n_1 \leq n_2$$

The ordering thus has height $h$.

We define $R \subset \mathcal{P}(A_m)$ to be the maximal subset of $\mathcal{P}(A_m)$ that contains no redundant elements. An element $S$ is *redundant* if:

$$\exists x, y \in S.\, x \sqsubseteq y \wedge x \neq y$$

In other words, $S$ is redundant if it contains two related elements of $A_m$, such that one subsumes the other.

An element $S \in R$ can be represented by an $n$-digit vector $u$, where $n = |L|$ and the digits are in the range $[0, h]$. The vector is defined as follows:

$$u(i) = \begin{cases} j & \text{if } (l_i, j) \in S, \\ 0 & \text{otherwise.} \end{cases}$$

The vector has a digit for each allocation site, and the value of the digit is the width of the abstract location in $S$ corresponding to the site, or 0 if there is none.

We use the following Hoare ordering on elements of $R$:

$$S_1 \sqsubseteq S_2 \Longleftrightarrow \forall x \in S_1.\ \exists y \in S_2.\ x \sqsubseteq y$$

The element $S_1$ is subsumed by $S_2$ if every element in $S_1$ is subsumed by some element in $S_2$. In the vector representation, the following is an equivalent ordering:

$$S_1 \sqsubseteq S_2 \Longleftrightarrow \forall i \in \{1, ..., |L|\}.\ u_1(i) \leq u_2(i)$$

In this representation, $S_1$ is subsumed by $S_2$ if each digit in $S_1$ is no more than the corresponding digit in $S_2$. The ordering relation induces a lattice with minimal element corresponding to $u_\perp(i) = 0$, and a maximal element corresponding to $u_\top(i) = h$. The maximal chain between $\perp$ and $\top$ is derived by increasing a single vector digit at a time by 1, so the chain, and therefore the lattice, has height $h \cdot |L| + 1$.

We now define a Galois connection between $\mathcal{P}(G)$ and $R$ as follows:

$$\begin{aligned} \gamma_m(S) &= \big\{(l, m', a) \bigm| (l, n) \in S \ \wedge \ \mathit{hier}(m, m') \leq n\big\} \\ \alpha_m(C) &= \sqcap\big\{S \bigm| C \sqsubseteq \gamma_m(S)\big\} \end{aligned}$$

The concretization of an abstract location $(l, n)$ with respect to machine $m$ is the set of all concrete locations with the same allocation site and located on machines that are at most $n$ away from $m$. The abstraction with respect to $m$ of a concrete location $(l, m', a)$ is an abstract location with the same allocation site and width equal to the distance between $m$ and $m'$.

Finally, we abstract the concrete domain $CS$ to the following abstract domain, which maps abstract locations and variables to *points-to sets* of abstract locations:

$$AS = (A_m + Id) \rightarrow R$$

An element $\sigma_A$ of $AS$ is subsumed by $\sigma'_A$ if the points-to set of each abstract location and variable in $\sigma_A$ is subsumed by the corresponding set in $\sigma'_A$. The elements of $AS$ are therefore ordered as follows:

$$\sigma_A \sqsubseteq \sigma'_A \Longleftrightarrow \forall x \in (A_m + Id).\ \sigma_A(x) \sqsubseteq \sigma'_A(x)$$

The resulting lattice has height in $\mathrm{O}(h \cdot |L| \cdot (|A_m| + |Id|)) = \mathrm{O}(h \cdot |L| \cdot (h \cdot |L| + |Id|))$. Since the number of allocation sites and identifiers is limited by the size of the input program $P$, the height is in $\mathrm{O}(h^2 \cdot |P|^2)$.

### 4.3 Abstract Semantics

For each expression in *Ti*, we provide inference rules for how the expression updates the abstract state $\sigma_A$. The judgments are of the form $\langle e, \sigma_A \rangle \Downarrow \langle S, \sigma'_A \rangle$, which means that expression $e$ in abstract state $\sigma_A$ can refer to the abstract locations $S$ and results in the modified abstract state $\sigma'_A$. As in §3.3, we use the notation $\sigma[g := v]$ to denote the function $\lambda x.\ \texttt{if}\ x = g\ \texttt{then}\ v\ \texttt{else}\ \sigma(x)$. Most of the rules are derived directly from the operational semantics of the language.

The rules for integer and variable expressions are straightforward. Neither updates the abstract state, and the latter returns the abstract locations in the points-to set of the variable.

$$\frac{}{\langle n, \sigma_A \rangle \Downarrow \langle \emptyset, \sigma_A \rangle} \qquad \frac{}{\langle x, \sigma_A \rangle \Downarrow \langle \sigma_A(x), \sigma_A \rangle}$$

An allocation returns the abstract location corresponding to the allocation site, with width 1.

$$\frac{}{\langle new_l\ \tau, \sigma_A \rangle \Downarrow \langle \{(l, 1)\}, \sigma_A \rangle}$$

The rule for dereferencing is similar to the operational semantics rule, except that all source abstract locations are simultaneously dereferenced.

$$\frac{\langle e, \sigma_A \rangle \Downarrow \langle S, \sigma'_A \rangle}{\langle *e, \sigma_A \rangle \Downarrow \langle \bigcup_{b \in S} \sigma'_A(b), \sigma'_A \rangle}$$

The rule for sequencing is also analogous to its operational semantics rule.

$$\frac{\langle e_1, \sigma_A \rangle \Downarrow \langle S_1, \sigma'_A \rangle \quad \langle e_2, \sigma'_A \rangle \Downarrow \langle S_2, \sigma''_A \rangle}{\langle e_1; e_2, \sigma_A \rangle \Downarrow \langle S_2, \sigma''_A \rangle}$$

The rule for variable assignment merely copies the source abstract locations into the points-to set of the target variable.

$$\frac{\langle e, \sigma_A \rangle \Downarrow \langle S, \sigma'_A \rangle}{\langle x := e, \sigma_A \rangle \Downarrow \langle S, \sigma'_A[x := \sigma'_A(x) \sqcup S] \rangle}$$

The type conversion expression can only succeed if the result is within the specified hierarchical distance, so it narrows all abstract locations that are outside that distance.

$$\frac{\langle e, \sigma_A \rangle \Downarrow \langle S, \sigma'_A \rangle}{\langle \texttt{convert}(e, n), \sigma_A \rangle \Downarrow \langle \{(l, min(k, n)) \mid (l, k) \in S\}, \sigma'_A \rangle}$$

The SPMD model of parallelism in *Ti* implies that the source expression of the `transmit` operation evaluates to abstract locations with the same labels on both the source and destination machines. The distance between the source and destination machines, however, is not statically known, so the resulting abstract locations must be assumed to have the maximum width.

$$\frac{\langle e_2, \sigma_A \rangle \Downarrow \langle S_2, \sigma'_A \rangle \quad \langle e_1, \sigma'_A \rangle \Downarrow \langle S_1, \sigma''_A \rangle}{\langle \texttt{transmit}\ e_1\ \texttt{from}\ e_2, \sigma_A \rangle \Downarrow \langle \{(l, h) \mid (l, m) \in S_1\}, \sigma''_A \rangle}$$

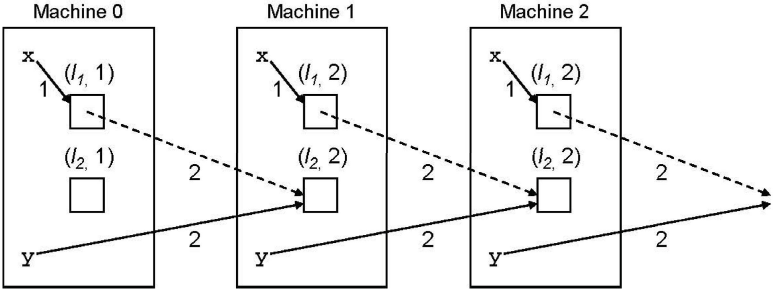


**Fig. 7.** The assignment $x \leftarrow y$ on machine 0 results in the abstract location $(l_2, 2)$ being added to the points-to set of $(l_1, 1)$, as shown by the first dashed arrow. The assignment on machine 1 results in the abstract location $(l_2, 2)$ being added to the points-to set of $(l_1, 2)$, as shown by the second dashed arrow. The assignment must also be accounted for on the rest of the machines. (Abstract locations in the figure are with respect to machine 0).

The rule for assignment through references is the most interesting. Suppose an abstract location $a_2 = (l_2, 2)$ is assigned into an abstract location $a_1 = (l_1, 1)$, as in Figure 7. Of course, we have to add $a_2$ to the points-to set of $a_1$. In addition, since *Ti* is SPMD, we have to account for the effect of the same assignment on a different machine. Consider the assignment on machine $m'$, where $hier(m, m') = 2$. The location $a_1$ relative to $m$ corresponds to a location $a'_1 = (l_1, 2)$ relative to $m'$. The location $a_2$ can correspond to a concrete location on $m'$, so its abstraction can be $a'_2 = (l_2, 1)$ relative to $m'$. But it can also correspond to a concrete location on $m''$ where $hier(m, m'') = hier(m', m'') = 2$, so its abstraction can also be $a''_2 = (l_2, 2)$. But since $a'_2 \sqsubseteq a''_2$, it is sufficient to assume that $a_2$ corresponds to $a''_2$ on $m'$. From the point of view of $m'$ then, the abstract location $(l_2, 2)$ should be added to the points-to set of the location $(l_1, 2)$.

In general, whenever an assignment occurs from $(l_2, n_2)$ to $(l_1, n_1)$, we have to update not only the points-to set of $(l_1, n_1)$ but the sets of all locations corresponding to label $l_1$ and of any width. As we show below, the proper update is to add the location $(l_2, max(n'_1, n_1, n_2))$ to the points-to set of each location $(l_1, n'_1)$. The rule is then

$$\frac{\langle e_1, \sigma_A\rangle \Downarrow \langle S_1, \sigma'_A\rangle \quad \langle e_2, \sigma'_A\rangle \Downarrow \langle S_2, \sigma''_A\rangle}{\langle e_1 \leftarrow e_2, \sigma_A\rangle \Downarrow \langle S_2, update(\sigma''_A, S_1, S_2)\rangle},$$

with $update$ defined as

$$\begin{aligned}
&update(\sigma, S_1, S_2) = \\
&\quad \lambda(l_1, n'_1) : L \times H \; . \\
&\qquad \sigma((l_1, n'_1)) \sqcup \big\{(l_2, max(n'_1, n_1, n_2)) \;\big|\; (l_1, n_1) \in S_1 \;\wedge\; (l_2, n_2) \in S_2\big\}.
\end{aligned}$$

### 4.4 Soundness

An abstract intepretation is sound if the abstraction and concretization functions are monotonic and form a Gallois connection, and the abstract inference rules for each operation is correct. The former condition was shown in §4.2.

Most of the abstract inference rules are derived directly from the operational semantics, so their correctness is obvious. The rule for assignment through a reference, however, is nontrivial, so we prove its correctness here.

Let $a_i^m$ represent the abstract location $a_i$ with respect to machine $m$. Let $n^m$ represent a width $n$ with respect to $m$.

Consider an assignment $e_1 \leftarrow e_2$. Let $m$ be the reference machine for the analysis. Without loss of generality, assume that $e_1$ evaluates to the lone abstract location $a_1^m = (l_1, n_1^m)$, and that $e_2$ evaluates to $a_2^m = (l_2, n_2^m)$. Consider the execution of this assignment on the following machines:

- On machines $m'$ such that $hier(m, m') \leq n_1^m$. This implies that the $(n_1^m - 1)$th ancestor of each $m'$ in the machine hierarchy is the same as that of $m$. As a result, abstract locations of width at least $n_1$ are the same with respect to both $m$ and $m'$. In particular, $a_1^{m'} = a_1^m$, so the assignment on any machine can target any concrete location in $a_1^m$.

  Now suppose $n_2^m < n_1^m$. Then the $a_2^{m'}$ are not equivalent for all machines $m'$. However, note that $a_2^{m'}$ contains the concrete locations $(l_2, m', a)$ for any $a$. Considering the assignment on all machines $m'$, the concrete locations in $a_1^m$ can receive any of the source concrete locations $(l_2, m', a)$ for all $m'$ and $a$. This set of source locations corresponds exactly to the abstract location $a_{2'}^m = (l_2, n_1^m)$.

  Suppose instead that $n_2^m \geq n_1^m$. Then the machines $m'$ all agree on the set $a_2^{m'} = a_2^m$. Thus, regardless of which machine the assignment is executed on, the source locations correspond exactly to $a_2^m$.

  In either case, any of the concrete locations corresponding to $a_1^m$ can now point to any of the concrete locations corresponding to $a_{2'}^m = (l_2, max(n_1^m, n_2^m))$. To capture this in the abstract inference, it is sufficient to add $a_{2'}^m$ to the points-to set of $a_1^m$. For consistency, $a_{2'}^m$ should also be added to the points-to set of any abstract location $a_{1'}^m \sqsubseteq a_1^m$, since any of the concrete locations corresponding to $a_{1'}^m$ can point to any of the concrete locations corresponding to $a_{2'}^m$.

  Thus, it is sufficient to add the abstract location $a_{2'}^m = (l_2, max(n_1^m, n_2^m))$ to the points-to set of any $a_{1'}^m = (l_1, n_{1'}^m)$ such that $n_{1'}^m \leq n_1^m$.
- On a machine $m'$, where $hier(m, m') > n_1^m$. The set of concrete locations corresponding to $a_1^{m'}$ all reside on machines a distance of $n_{1'}^m = hier(m, m')$ away from machine $m$. Thus, $a_1^{m'} \sqsubseteq a_{1'}^m$, where $a_{1'}^m = (l_1, n_{1'}^m)$.

  Now suppose $n_2^m < n_{1'}^m$. Then all the concrete locations corresponding to $a_2^{m'}$ reside at a distance of $n_{1'}^m$ from machine $m$, so that $a_2^{m'} \sqsubseteq a_{2'}^m$, where $a_{2'}^m = (l_2, n_{1'}^m)$. Thus, the source locations can be soundly approximated by $a_{2'}^m$.

  Suppose instead that $n_2^m \geq n_{1'}^m$. Then $m$ and $m'$ agree on $a_2^{m'} = a_2^m$, so the source locations correspond to $a_2^m$.

  In either case, some of the concrete locations corresponding to $a_{1'}^m$ can now point to some of the concrete locations corresponding to $a_{2'}^m = (l_2, max(n_{1'}^m, n_2^m))$. Soundness can be maintained, though precision lost, if the analysis assumes that any concrete location corresponding to $a_{1'}^m$ can point to any concrete location corresponding to $a_{2'}^m$. Thus, $a_{2'}^m$ should be added to the points-to set of $a_{1'}^m$.

  Now consider an abstract location $a_{1''}^m = (l_1, n_{1''}^m)$, where $n_{1''}^m < n_{1'}^m$. All concrete locations represented by $a_{1''}^m$ reside less than a distance of $n_{1'}^m$ away from

$m$. Since all concrete locations corresponding to $a_1^{m'}$ reside at a distance of $n_{1'}^m$ from $m$, the abstract locations $a_{1''}^m$ and $a_1^{m'}$ do not intersect. Thus, none of the concrete locations in $a_{1''}^m$ are targeted by the assignment, so its points-to set does not need to be updated.

Thus, it is sufficient to add the abstract location $a_{2'}^m = (l_2, max(n_{1'}^m, n_2^m))$ to the points-to set of each $a_{1'}^m = (l_1, n_{1'}^m)$ such that $n_{1'}^m > n_1^m$.

Summarizing over all possibilities, we obtain the rule that the abstract location $a_{2'}^m = (l_2, max(n_{1'}^m, n_1^m, n_2^m))$ is to be added to the points-to set of any $a_{1'}^m = (l_1, n_{1'}^m)$. This corresponds exactly to the update rule provided in §4.3.

### 4.5 Algorithm

The set of inference rules, instantiated over all the expressions in a program and applied in some arbitrary order[2], composes a function $F : AS \rightarrow AS$. Only the two assignment rules affect the input state $\sigma_A$, and in both rules, the output consists of a least upper bound operation involving the input state. As a result, $F$ is a monotonically increasing function, and the least fixed point of $F$, $F_0 = \sqcup_n F^n(\lambda x.\ \emptyset)$, is the analysis result.

The function $F$ has a rule for each program expression, so it takes time in $\mathrm{O}(|P|)$ to apply it[3], where $P$ is the input program. Since the lattice over $AS$ has height in $\mathrm{O}(h^2 \cdot |P|^2)$, it takes time in $\mathrm{O}(h^2 \cdot |P|^3)$ to compute the fixed point of $F$.

In our implementation, we have found that the running time of the analysis varies little between one, two, and three levels of hierarchy. For the benchmarks in §5, a three-level analysis takes no more than 10% longer than a single-level analysis and less than 5% longer than a two-level analysis. Thus, the three-level analysis is far more efficient than running a two-level analysis twice.

## 5 Evaluation

The pointer information computed in §4 can be applied to multiple analyses and optimizations for parallel programs. We evaluate the pointer analysis by using it for race detection. In [13], we apply it as well to enforcement of sequential consistency and describe how it can be used to infer data locality and privacy.

We use the following set of benchmarks:

- **`amr`** [27] (7581 lines) Chombo adaptive mesh refinement suite [3] in Titanium.
- **`gas`** [5] (8841 lines): Hyperbolic solver for a gas dynamics problem in computational fluid dynamics.
- **`ft`** [9] (1192 lines): NAS Fourier transform benchmark [4] in Titanium.
- **`cg`** [9] (1595 lines): NAS conjugate gradient benchmark [4] in Titanium.
- **`mg`** [9] (1952 lines): NAS multigrid benchmark [4] in Titanium.

[2] Since the analysis is flow-insensitive, the order of application is not important.

[3] We ignore the cost of the join operations here. In practice, points-to sets tend to be small, so the cost of joining them can be neglected.

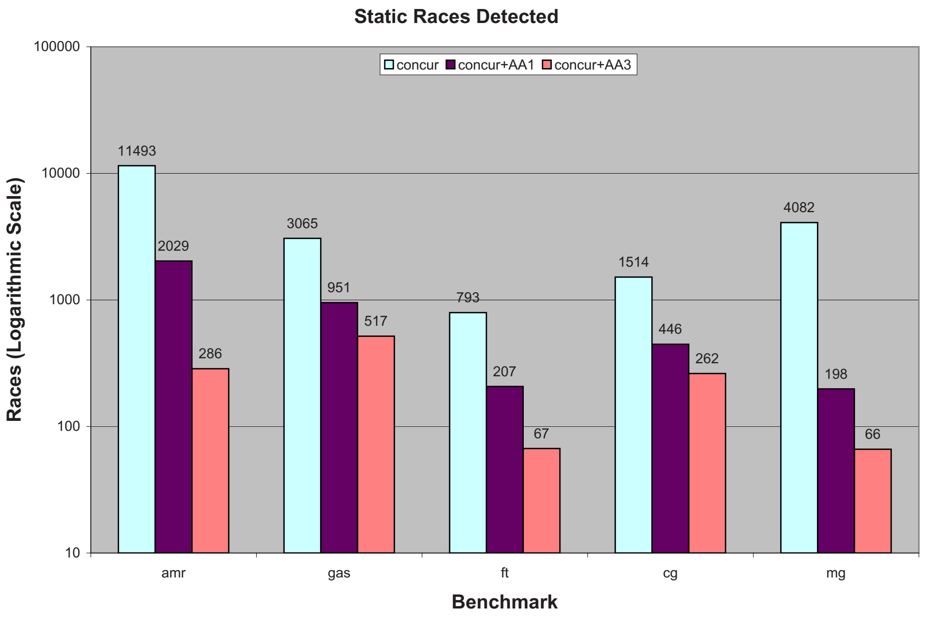


**Fig. 8.** Number of data races reported for different levels of analysis

The line counts for the above benchmarks underestimate the amount of code actually analyzed, since all reachable code in the 37,000 line Titanium and Java 1.0 libraries is also processed.

A race condition occurs when two memory accesses can occur simultaneously on different threads, they can be to the same memory location, and at least one is a write. An existing concurrency analysis for Titanium [15] can conservatively determine which accesses are simultaneous. The pointer analysis can detect if two accesses may be to the same location by checking if they can operate on abstract locations whose concretizations with respect to different machines overlap. In a single-level analysis, all abstract locations with the same label overlap, while in a multi-level analysis, they do not overlap if they are both machine-local (i.e. have width 1). Thus, a multi-level analysis results in higher race detection precision than a single-level analysis.

Static information is generally not enough to determine with certainty that two memory accesses compose a race, so nearly all reported races are false positives. (The correctness of the concurrency and pointer analyses ensure that no false negatives occur.) We therefore consider a race detector that reports the fewest races to be the most effective

Figure 8 compares the effectiveness of three levels of race detection:

- **concur:** Our concurrency analysis[4] [15] is used to eliminate non-concurrent memory accesses. Sharing inference [17] is used to eliminate accesses to thread-private data.

[4] The most precise analysis in [15] is used, which was labeled as *feasible* in that paper.

- **concur+AA1:** A single-level pointer analysis is added to eliminate false aliases.
- **concur+AA3:** A three-level pointer analysis is added to eliminate false aliases.

The results show that the pointer analysis can eliminate most of the races reported by our detector. The addition of pointer analysis removes most of the races discovered by only using the concurrency analysis, with a three-level analysis providing significant benefits over a one-level analysis. However, the results are still not precise enough for production use. The pointer analysis does not currently distinguish between array indices, and since Titanium programs tend to make extensive use of arrays in their data structures, this results in a significant number of false aliases. However, the addition of an array index analysis [20,19,18,22] should remove most of these false aliases, and consequently most of the false positives reported by the race detector.

## 6 Related Work

The language and type system we presented here are generalizations of those described by Liblit and Aiken [16]. They defined a two-level hierarchy and used it to produce a constraint-based analysis that infers locality information about pointers. Later with Yelick, they extended the language and type system to consider sharing of data, and they defined another constraint-based analysis to infer sharing properties of pointers [17].

Pointer analysis was first described by Andersen [2], and later extended by others to parallel programs. Rugina and Rinard developed a thread-aware alias analysis for the Cilk multithreaded programming language [23] that is both flow-sensitive and context-sensitive. Others such as Zhu and Hendren [29] and Hicks [11] have developed flow-insensitive versions for multithreaded languages. However, none of these analyses consider hierarchical, distributed machines.

The pointer analysis we presented here is a generalization and formalization of the analysis sketched in a previous paper [14]. That analysis is similar to a two-level version of our hierarchical analysis, but the abstraction is quite different. Only the abstraction of the `transmit` operation was described in that paper, though an almost complete implementation was done.

## 7 Conclusion

In this paper, we introduced a program analysis technique for pointers, which has applications in detecting program errors and enabling optimizations. The novelty of the analysis derives from its view of the machine as an arbitrary hierarchy of processors, with the analysis proving that the range of a pointer is limited to a given hierarchy.

Our analysis was presented on a small language, *Ti*, which decouples the analysis from specifics of the language. The type system allows for references of different widths, corresponding to local and global pointers in PGAS languages. We demonstrated the analysis with an implementation in the Titanium language, a global address space language with three levels of hierarchy. Our results show that the multi-level analysis is significantly more accurate than one based on only a single level.

There are several potential clients of our analysis, and in this paper we presented one such client, a static race detection algorithm, which combined the pointer analysis with our existing concurrency analysis to detect races in Titanium programs. Even on relatively complicated benchmarks codes, our results show that the more accurate pointer analysis has a significant impact on the quality of the race analysis. Our results indicate the value of exposing the hierarchy within the language and compiler to balance the desire of programmers for both simplicity and high performance.

## References

1. Allen, E., Chase, D., Luchangco, V., Maessen, J.-W., Ryu, S., G. L. S. Jr., Tobin-Hochstadt, S.: The Fortress Language Specification, Version 0.866. Sun Microsystem Inc. (February 2006)
2. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (May 1994)
3. Applied Numerical Algorithms Group (ANAG). Chombo, `http://seesar.lbl.gov/ANAG/software.html`
4. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, D., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS Parallel Benchmarks. The International Journal of Supercomputer Applications 5(3), 63–73 (1991)
5. Berger, M., Colella, P.: Local adaptive mesh refinement for shock hydrodynamics. Journal of Computational Physics 82(1), 64–84 (1989) (Lawrence Livermore Laboratory Report No. UCRL-97196)
6. Bonachea, D.: GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, University of California, Berkeley (November 2002)
7. Carlson, W., Draper, J., Culler, D., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences (1999)
8. Cray Inc. Chapel Specification 0.4 (February 2005)
9. Datta, K., Bonachea, D., Yelick, K.: Titanium performance and potential: an NPB experimental study. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339, Springer, Heidelberg (2006)
10. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 2nd edn. Addison-Wesley, London, UK (2000)
11. Hicks, J.: Experiences with compiler-directed storage reclamation. In: FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture, New York, USA, pp. 95–105. ACM Press, New York, USA (1993)
12. Hilfinger, P.N., Bonachea, D., Gay, D., Graham, S., Liblit, B., Pike, G., Yelick, K.: Titanium language reference manual. Technical Report UCB/CSD-04-1163-x, University of California, Berkeley (September 2004)
13. Kamil, A.: Analysis of Partitioned Global Address Space Programs. Master's thesis, University of California, Berkeley (December 2006)
14. Kamil, A., Su., J., Yelick, K.: Making sequential consistency practical in Titanium. In: Supercomputing 2005 (November 2005)
15. Kamil, A., Yelick, K.: Concurrency analysis for parallel programs with textually aligned barriers. In: Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (October 2005)

16. Liblit, B., Aiken, A.: Type systems for distributed data structures. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 2000, ACM Press, New York (2000)
17. Liblit, B., Aiken, A., Yelick, K.: Type systems for distributed data sharing. In: International Static Analysis Symposium, San Diego, California (June 2003)
18. Lin, Y., Padua, D.A.: Analysis of irregular single-indexed array accesses and its applications in compiler optimizations. In: CC '00: Proceedings of the 9th International Conference on Compiler Construction, London, UK, pp. 202–218. Springer, Heidelberg (2000)
19. Maydan, D.E., Amarasinghe, S.P., Lam, M.S.: Array-data flow analysis and its use in array privatization. In: POPL '93. Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, pp. 2–15. ACM Press, New York, NY, USA (1993)
20. Maydan, D.E., Amarsinghe, S., Lam, M.S.: Data dependence and data-flow analysis of arrays. In: Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, London, UK, pp. 434–448. Springer, Heidelberg (1993)
21. Netzer, R.H.B., Miller, B.P.: What are race conditions?: Some issues and formalizations. ACM Lett. Program. Lang. Syst. 1(1), 74–88 (1992)
22. Paek, Y., Hoeflinger, J., Padua, D.: Efficient and precise array access analysis. ACM Trans. Program. Lang. Syst. 24(1), 65–109 (2002)
23. Rugina, R., Rinard, M.: Pointer analysis for multithreaded programs. In: PLDI '99.: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, New York, NY, USA, pp. 77–90. ACM Press, New York, USA (1999)
24. Saraswat, V.: Report on the Experimental Language X10, Version 0.41. IBM Research (February 2006)
25. Silicon Graphics. CF90 co-array programming manual. Technical Report SR-3908 3.1, Cray Computer (1994)
26. The UPC Consortium. UPC Language Specifications, Version 1.2 (May 2005)
27. Wen, T., Colella, P.: Adaptive mesh refinement in titanium. In: Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS) (2005)
28. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performance Java dialect. In: Workshop on Java for High-Performance Network Computing, Stanford, California (February 1998)
29. Zhu, Y., Hendren, L.J.: Communication optimizations for parallel C programs. In: PLDI '98. Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, New York, NY, USA, pp. 199–211. ACM Press, New York, USA (1998)

# Semantics-Based Transformation of Arithmetic Expressions

Matthieu Martel

CEA - LIST
CEA F91191 Gif-Sur-Yvette Cedex, France
matthieu.martel@cea.fr

**Abstract.** Floating-point arithmetic is an important source of errors in programs because of the loss of precision arising during a computation. Unfortunately, this arithmetic is not intuitive (e.g. many elementary operations are not associative, inversible, etc.) making the debugging phase very difficult and empiric.

This article introduces a new kind of program transformation in order to automatically improve the accuracy of floating-point computations. We use P. Cousot and R. Cousot's framework for semantics program transformation and we propose an offline transformation. This technique was implemented, and the first experimental results are presented.

## 1 Introduction

In this article, we introduce a new kind of program transformation in order to improve the precision of the evaluation of expressions in floating-point arithmetic. We consider that an expression implements a formula obeying the usual laws of mathematics. This means that, in particular, the evaluation of the formula in infinite precision yields an exact result and that algebraic rules like associativity, commutativity or distributivity do not modify the meaning of a formula. However, floating-point arithmetic differs strongly from real number arithmetic: the values have a finite number of digits and the algebraic laws mentioned earlier no longer hold. Consequently, the evaluation by a computer of mathematically equivalent formulas (for example $x \times (1 + x)$ and $x + x^2$) possibly leads to very different results.

Our work is motivated by the fact that, in programs, errors due to floating-point arithmetic are very difficult to understand and to rectify. Recently, validation techniques based on abstract interpretation have been developed to assert the numerical accuracy of these calculations [13,8] but, while these tools enable one to detect the imprecisions and, possibly, to understand their origin, they do not help the programmer to correct the programs. Unfortunately, floating-point arithmetic is not intuitive, making the debugging phase very difficult and empiric: there exists no methodology to improve the accuracy of a computation and we have at most a set of tricks like "sort numbers increasingly before adding

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 298–314, 2007.

them" or "use Horner's method to evaluate a polynomial." Performing these transformations by hand is tedious because the computer arithmetic is subtle. Therefore, their automatization is of great practical interest. Even if static analysis techniques have already given rise to industrially usable tools to assert the numerical precision of critical codes [8,9], there is an important gap between validation and automatic correction. To our knowledge, this article is the first attempt in that new direction.

We introduce a new kind of program transformation, in order to automatically improve the "quality" of an arithmetic expression with respect to some evaluation criterion: the precision of floating-point computations. We use P. Cousot and R. Cousot's framework for semantics program transformation [6] by abstract interpretation [5] and we propose an offline transformation. The methodology of [6] enables us to define a semantics transformation that would be far more difficult to obtain at the syntactic level, since there is no strong syntactic relation between the source and transformed expressions.

For the sake of simplicity, we restrict ourselves to arithmetic expressions, neglecting, in this first work, the statements of a full programming language. However, our techniques are not specific to expressions and can be extended to complete programming languages.

The main steps of our method are the following. First, we introduce a non-deterministic small-step operational semantics for the evaluation of real expressions. Basically, algebraic laws like associativity, commutativity or distributivity make it possible to evaluate the same expression in many different ways (all confluent to the same final result.) Next, the same semantics is applied to floating-point arithmetic. In this case, different evaluations of an expression yield different results because the algebraic laws of the reals do not work any longer. Then we compute the quality of each execution path of the floating-point arithmetic based semantics by means of a non-standard domain (e.g. the global error arithmetic developed for validation of floating-point computation [13,12]). However, because there are too many paths in the previous semantics, we define a new abstract semantics in which sets of traces are merged into abstract traces. Basically, we merge traces in which sub-expressions have been evaluated approximatively in the same way, using abstract expressions of limited height. The semantics transformation then consists of computing (approximatively) the execution path which optimizes the quality of the evaluation. The correctness of the transformation stems from the fact that, at the observational level (i.e. in the reals), all the execution paths that we consider lead to the same final result. Other classical abstractions of sets of numbers by intervals is used, in order to deal with sets of values and to find the best expression for a range of inputs. A prototype has been implemented and we also present some experimental results.

The rest of this article is organized as follows: Section 2 gives an overview of our transformation and of the semantics we use. Section 3 and Section 4 introduce the concrete and abstract semantics. The transformation is presented in Section 5 and experimental results are given in Section 6. Sections 7 and 8 are dedicated to perspectives and concluding remarks.

## 2 Overview

As mentioned in the introduction, we aim at transforming mathematic expressions in order to improve the precision of their evaluation in floating-point arithmetic. For example, let us consider the simple formula which computes the area of a rectangular parallelepiped of dimension $a \times b \times c$:

$$A = 2 \times \big((a \times b) + (b \times c) + (c \times a)\big) \tag{1}$$

Let us consider a thin parallelepiped of dimensions $a = 1 \quad b = c = \frac{1}{9}$. With these values, the examination of Equation (1) reveals that $ab \gg ac$ and $ab \gg bc$. It is well-known that in floating-point arithmetic, adding numbers of different magnitude may lead to important precision loss: if $x \ll y$ then, possibly, $x +_{\mathbb{F}} y = y$ (this is called an absorption). In our example, absorptions arise in the direct evaluation of $A$. The transformation introduced in this article enables one to automatically rewrite the original expression (where $d = 2$):

```
d*(((a*b)+(b*c))+(c*a))
```

into the new expression:

```
(((c*a)*d)+(d*(b*c)))+(d*(a*b))
```

In this new formula, the smallest terms are summed first. Furthermore, the product is distributed and this avoids a multiplication of roundoff errors of the additions by a large value. This transformation relies on several semantics which are summarized below.

- $\rightarrow_{\mathbb{F}}$ is the concrete semantics based on the floating-point arithmetic. This semantics corresponds to the evaluation of an expression $e$ by a computer. $\rightarrow_{\mathbb{F}}$ is defined in Section 3.1.
- $\rightarrow_{\mathbb{R}}$ is the concrete semantics based on real number arithmetic. In $\mathbb{R}$, algebraic rules hold, like associativity, distributivity, etc. $\rightarrow_{\mathbb{R}}$ is defined in Section 3.2.
- $\rightarrow_{\mathbb{E}}$ is the non-standard semantics based on the arithmetic of floating-point numbers with global errors. This semantics calculates the exact global error between a real computation and a floating-point computation [12].$\rightarrow_{\mathbb{E}}$ is defined in Section 3.3.
- $\longrightarrow$ is the non-standard semantics used to define our abstract interpretation. $\longrightarrow$ is defined in Section 4.1.
- $\xrightarrow{A}_k$ is the abstract semantics. $k$ is a parameter which defines the precision of the semantics. $\xrightarrow{A}_k$ is defined in Section 4.2.

## 3 Concrete Semantics of Expressions

In this section, we introduce some concrete semantics of expressions, for floating-point arithmetic, for real arithmetic and for floating-point numbers with global errors (the semantics $\rightarrow_{\mathbb{F}}$, $\rightarrow_{\mathbb{R}}$ and $\rightarrow_{\mathbb{E}}$ mentioned in Section 2). $\rightarrow_{\mathbb{F}}$ is the semantics used by a computer which complies with the IEEE754 Standard [1], $\rightarrow_{\mathbb{R}}$

$$\begin{array}{ccc}
\dfrac{v = v_1 +_{\mathbb{F}} v_2}{v_1 + v_2 \ \rightarrow_{\mathbb{F}} \ v} & \dfrac{e_1 \ \rightarrow_{\mathbb{F}} \ e_1'}{e_1 + e_2 \ \rightarrow_{\mathbb{F}} \ e_1' + e_2} & \dfrac{e_2 \ \rightarrow_{\mathbb{F}} \ e_2'}{v_1 + e_2 \ \rightarrow_{\mathbb{F}} \ v_1 + e_2'} \\
\dfrac{v = v_1 \times_{\mathbb{F}} v_2}{v_1 \times v_2 \ \rightarrow_{\mathbb{F}} \ v} & \dfrac{e_1 \ \rightarrow_{\mathbb{F}} \ e_1'}{e_1 \times e_2 \ \rightarrow_{\mathbb{F}} \ e_1' \times e_2} & \dfrac{e_2 \ \rightarrow_{\mathbb{F}} \ e_2'}{v_1 \times e_2 \ \rightarrow_{\mathbb{F}} \ v_1 \times e_2'}
\end{array}$$

**Fig. 1.** The reduction rules for floating point arithmetic

is used in the correctness proofs where it plays the role of observer [6], and $\rightarrow_{\mathbb{E}}$ is used to define non-standard and abstract semantics of programs.

For the sake of simplicity, we only consider elementary arithmetic expressions generated by the grammar:

$$e \ ::= \ v \mid x \mid e_1 + e_2 \mid e_1 \times e_2. \tag{2}$$

In Equation (2), $v$ denotes a value and $x \in \mathrm{Id}$ is a constant whose value is given by a global environment. These global variables are implemented in our prototype, and they introduce no theoretical difficulty. We omit them in all the formal semantics.

### 3.1 Floating-Point Arithmetic Based Semantics

The semantics $\rightarrow_{\mathbb{F}}$ just defines how an expression is evaluated by a computer, following the IEEE754 Standard for floating-point arithmetic.

Let $\uparrow_\circ : \mathbb{R} \rightarrow \mathbb{F}$ be the function which returns the roundoff of a real number following the rounding mode $\circ \in \{\circ_{-\infty}, \circ_{+\infty}, \circ_0, \circ_\sim\}$ [1]. $\uparrow_\circ$ is fully specified by the IEE754 Standard which also requires, for any elementary operation $\Diamond$, that:

$$x_1 \ \Diamond_{\mathbb{F},\circ} \ x_2 \ = \uparrow_\circ (x_1 \ \Diamond_{\mathbb{R}} \ x_2) \tag{3}$$

Equation (3) states that the result of an operation between floating-point numbers is the roundoff of the exact result of this operation. In this article, we also use the function $\downarrow_\circ$: $\mathbb{R} \rightarrow \mathbb{R}$ which returns the roundoff error. We have $\downarrow_\circ (r) = r - \uparrow_\circ (r)$.

The floating-point arithmetic based semantics of expressions is defined by the rules of Figure 1. This semantics is obvious but we will need it to prove the correctness of the transformation in Section 4.3.

### 3.2 Real Arithmetic Based Semantics

The exact evaluation of the expressions in Equation (2) is given by the real arithmetic. So, we define the reduction rules $\rightarrow_{\mathbb{R}}$ by assuming that any value $v$ belongs to $\mathbb{R}$ and by using the reduction rules of Figure 2 in which $\oplus$ and $\otimes$ stand for $+_{\mathbb{R}}$ and $\times_{\mathbb{R}}$ (the addition and product between real numbers).

The rules of equations (4) to (7) are straightforward. The rule of Equation (8) relies on the syntactic relation $\equiv$ defined as being the smallest equivalence relation containing relations $(i)$ to $(vii)$ of Figure 2. The equivalence $\equiv$ identifies

$$\frac{v = v_1 \oplus v_2}{v_1 + v_2 \;\rightarrow\; v} \tag{4}$$

$$\frac{v = v_1 \otimes v_2}{v_1 \times v_2 \;\rightarrow\; v} \tag{5}$$

$$\frac{e_1 \;\rightarrow\; e_1'}{e_1 + e_2 \;\rightarrow\; e_1' + e_2} \tag{6}$$

$$\frac{e_1 \;\rightarrow\; e_1'}{e_1 \times e_2 \;\rightarrow\; e_1' \times e_2} \tag{7}$$

$$\frac{e \equiv e_1 \quad e_1 \;\rightarrow\; e_1' \quad e_1' \equiv e'}{e \;\rightarrow\; e'} \tag{8}$$

$$\begin{array}{rl}
\textit{(i)} & (e_1 + e_2) + e_3 \equiv e_1 + (e_2 + e_3) \\
\textit{(ii)} & e_1 + e_2 \equiv e_2 + e_1 \\
\textit{(iii)} & e \equiv e + 0 \\
\textit{(iv)} & (e_1 \times e_2) \times e_3 \equiv e_1 \times (e_2 \times e_3) \\
\textit{(v)} & e_1 \times e_2 \equiv e_2 \times e_1 \\
\textit{(vi)} & e \equiv e \times 1 \\
\textit{(vii)} & e_1 \times (e_2 + e_3) \equiv e_1 \times e_2 + e_1 \times e_3
\end{array}$$

**Fig. 2.** The reduction rules for arithmetic expressions

arithmetic expressions which are equal in the reals, using associativity, distributivity and the neutral elements of $\mathbb{R}$. Equation (8) makes our transition system non-deterministic: there exist many reduction paths to evaluate the same expression. However, in $\mathbb{R}$, this transition system is (weakly) confluent and all the evaluations yield the same final result. This is summed up by the following property in which $\rightarrow_{\mathbb{R}}^*$ denotes the transitive closure of $\rightarrow_{\mathbb{R}}$.

**Property 1.** *Let $e$ be an arithmetic expression. If $e \rightarrow_{\mathbb{R}} e_1$ and $e \rightarrow_{\mathbb{R}} e_2$ then there exists $e'$ such that $e_1 \rightarrow_{\mathbb{R}}^* e'$ and $e_2 \rightarrow_{\mathbb{R}}^* e'$.*

### 3.3 Global Error Semantics

To define the global error semantics $\rightarrow_{\mathbb{E}}$, we first introduce the domain $\mathbb{E} = \mathbb{F} \times \mathbb{R}$. Intuitively, in a value $(x, \mu) \in \mathbb{E}$, $\mu$ measures the distance between the floating-point result of a computation $x$ and the exact result. The elements of $\mathbb{E}$ are ordered by $(x_1, \mu_1) \prec (x_2, \mu_2) \iff \mu_1 \leq \mu_2$.

Formally, a value $v$ is denoted by a pair $(x, \mu)$ where $x \in \mathbb{F}$ denotes the floating-point number used by the computer and $\mu \in \mathbb{R}$ denotes the exact error attached to $x$. For example, in simple precision, the real number $\frac{1}{3}$ is represented by the value $x = (\uparrow_\circ (\frac{1}{3}), \downarrow_\circ (\frac{1}{3})) = (0.333333, (\frac{1}{3} - 0.333333))$. The semantics interprets a constant $c$ by $(\uparrow_\circ (c), \downarrow_\circ (c))$ and, for $v_1 = (x_1, \mu_1)$ and $v_2 = (x_2, \mu_2)$, the operations are defined by:

$$v_1 +_{\mathbb{E}} v_2 = (\uparrow_\circ (x_1 +_{\mathbb{R}} x_2), [\mu_1 + \mu_2 + \downarrow_\circ (x_1 +_{\mathbb{R}} x_2)]) , \tag{9}$$

$$v_1 \times_{\mathbb{E}} v_2 = (\uparrow_\circ (x_1 \times_{\mathbb{R}} x_2), [\mu_1 x_2 +_{\mathbb{R}} \mu_2 x_1 +_{\mathbb{R}} \mu_1\mu_2 +_{\mathbb{R}} \downarrow_\circ (x_1 \times_{\mathbb{R}} x_2)])\,. \qquad (10)$$

The global semantics $\rightarrow_{\mathbb{E}}$ is defined by the reduction rules of equations (4) to (8) of Figure 2 and by the domain $\mathbb{E}$ for the values. The operators $\oplus$ and $\otimes$ are the addition $+_{\mathbb{E}}$ and the product $\times_{\mathbb{E}}$.

Similarly to the semantics $\rightarrow_{\mathbb{R}}$ of Section 3.2, $\rightarrow_{\mathbb{E}}$ is non-deterministic since it also uses the rule of Equation (8) based on the syntactic relation $\equiv$. However, in $\mathbb{E}$, the operations are neither associative nor distributive and the reduction paths no longer are confluent.

**Remark 2.** *In general, for an arithmetic expression $e$, there exist reduction steps $e \rightarrow_{\mathbb{E}} e_1$ and $e \rightarrow_{\mathbb{E}} e_2$ such that there exists no expression $e'$ such that $e_1 \rightarrow^*_{\mathbb{E}} e'$ and $e_2 \rightarrow^*_{\mathbb{E}} e'$.*

Nonetheless, the arithmetic $\mathbb{E}$ provides a way to compare the different execution paths of $\rightarrow_{\mathbb{E}}$ using the error measure $\mu$ attached to each value. We may consider a path $e \rightarrow^*_{\mathbb{E}} v_1$ is *better* than another path $e \rightarrow^*_{\mathbb{E}} v_2$ if $v_1 \prec v_2$. The code transformation introduced in the following sections consists of building a new arithmetic expression from the minimal trace corresponding to the evaluation of an expression $e$. But because there are possibly an exponential number of traces corresponding to the evaluation of $e$, we first merge some of them into abstract traces. The transformation is then based on the minimal abstract trace.

## 4 Abstract Semantics

The abstract semantics $\xrightarrow{A}_k$, introduced in Section 4.2, relies on the non-standard semantics $\longrightarrow$ of Section 4.1. In Section 4.3, we prove the correctness of the abstraction.

### 4.1 Non-standard Semantics

Basically, the non-standard semantics records, during a computation, how each intermediary result (sub-expression reduced to a value) was obtained. A label $\ell \in \mathcal{L}$ is attached to each value occurring in the expressions and we use two environments: The function $\rho : \mathcal{L} \rightarrow$Expr maps any label $\ell$ to the expression $e$ whose evaluation has lead to $v^\ell$. The environment $\sigma \;:\;$ Expr $\rightarrow \mathbb{E}$ maps expressions to the result of their evaluation in the domain $\mathbb{E}$. We let $\mathrm{Env}_\rho$ and $\mathrm{Env}_\sigma$ denote the sets of such environments. This information is useful in the abstract semantics of Section 4.2.

Initially, a unique label is attached to each value occurring in an expression and a fresh label is associated to the result of each operation. For example, assuming that initially $\rho(\ell_1) = 1^{\ell_1}$, $\rho(\ell_2) = 2^{\ell_2}$ and $\rho(\ell_3) = 3^{\ell_3}$, the expression $(1^{\ell_1} + (2^{\ell_2} + 3^{\ell_3}))$ is evaluated as follows in the non-standard semantics:

$$\langle \rho, \sigma, (1^{\ell_1} + (2^{\ell_2} + 3^{\ell_3}))\rangle \rightarrow \langle \rho', \sigma', 1^{\ell_1} + 5^{\ell_4}\rangle \rightarrow \langle \rho'', \sigma'', 6^{\ell_5}\rangle$$

$$\frac{v = v_1 +_{\mathbb{E}} v_2 \quad \ell \notin \mathrm{Dom}(\rho)}{\langle \rho, \sigma, v_0^{\ell_0} + v_1^{\ell_1}\rangle \longrightarrow \langle \rho[\ell \mapsto \rho(\ell_1) + \rho(\ell_2)], \sigma[\rho(\ell_1) + \rho(\ell_2) \mapsto v], v^{\ell}\rangle} \tag{11}$$

$$\frac{v = v_1 \times_{\mathbb{E}} v_2 \quad \ell \notin \mathrm{Dom}(\rho)}{\langle \rho, \sigma, v_0^{\ell_0} \times v_1^{\ell_1}\rangle \longrightarrow \langle \rho[\ell \mapsto \rho(v_1^{\ell_1}) \times \rho(v_2^{\ell_2})], \sigma[\rho(\ell_1) \times \rho(\ell_2) \mapsto v], v^{\ell}\rangle} \tag{12}$$

$$\frac{\langle \rho, \sigma, e_0\rangle \longrightarrow \langle \rho', \sigma', e_2\rangle}{\langle \rho, \sigma, e_0 + e_1\rangle \longrightarrow \langle \rho', \sigma', e_2 + e_1\rangle} \tag{13}$$

$$\frac{\langle \rho, \sigma, e_0\rangle \longrightarrow \langle \rho', \sigma', e_2\rangle}{\langle \rho, \sigma, e_0 \times e_1\rangle \longrightarrow \langle \rho', \sigma', e_2 \times e_1\rangle} \tag{14}$$

$$\frac{e \equiv e_1 \qquad \langle \rho, \sigma, e_1\rangle \longrightarrow \langle \rho', \sigma', e_2\rangle \qquad e_2 \equiv e_3}{\langle \rho, \sigma, e_0\rangle \longrightarrow \langle \rho', \sigma', e_3\rangle} \tag{15}$$

**Fig. 3.** The non-standard semantics

where $\rho' = \rho[\ell_4 \mapsto 2^{\ell_2} + 3^{\ell_3}]$, $\sigma' = \sigma[2^{\ell_2} + 3^{\ell_3} \mapsto 5]$, $\rho'' = \rho'[\ell_5 \mapsto 1^{\ell_1} + (2^{\ell_2} + 3^{\ell_3})]$ and $s'' = \sigma'[1^{\ell_1} + (2^{\ell_2} + 3^{\ell_3}) \mapsto 6]$. In this example, for the sake of simplicity, values are integers instead of values of $\mathbb{E}$.

The non-standard semantics is given in Figure 3. We assume that, initially, $\rho(\ell) = v^{\ell}$ for any value $v^{\ell}$ occurring in the expression. Equations (11) and (12) respectively perform an addition and a product in $\mathbb{E}$. A new label $\ell$ is assigned to the result $v$ of the operation and the environment $\rho$ is extended in order to relate $\ell$ to the expression which has been evaluated. Similarly, $\sigma$ is extended in order to record the result of the evaluation of the expression. The other rules only differ from the rules of the concrete semantics in that they propagate the environments $\rho$ and $\sigma$.

### 4.2 Abstract Semantics

In order to decrease the size of the non-standard semantics, the abstract semantics merges traces in which sub-expressions have been evaluated approximatively in the same way. More precisely, instead of the environments $\rho$ and $\sigma$, we use abstract environments $\rho^{\sharp}$ mapping labels to abstract expressions of limited height and abstract environments $\sigma^{\sharp}$ mapping expressions of limited height to unions of values. Next, we merge the paths in which sub-expressions have been evaluated almost in the same way, i.e. by the same abstract expressions.

From a formal point of view, the set $\mathrm{Expr}_k^{\sharp}$ of abstract expressions of height at most $k$ is recursively defined by:

$$\begin{aligned} \eta_0 &::= v^{\sharp\ell} \mid \top_{\eta} \\ \eta_k &::= \eta_{k-1} \mid \eta_{k-1} + \eta_{k-1} \mid \eta_{k-1} \times \eta_{k-1}. \end{aligned} \tag{16}$$

The values occurring in the abstract expressions belong to the abstract domain $\mathbb{E}^{\sharp}$. Let $\wp(X)$ denote the powerset of $X$. Abstract and concrete floating-point numbers with errors are related by the Galois connection

$$\langle \wp(\mathbb{E}), \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{E}^{\sharp}, \sqsubseteq^{\sharp}_{\mathbb{E}}\rangle.$$

$$\frac{v^\sharp = \bigcup_{\substack{\eta_1 \in \rho^\sharp(\ell_1)\\ \eta_2 \in \rho^\sharp(\ell_2)}} \sigma^\sharp(\eta_1) +^\sharp_{\mathbb{E}} \sigma^\sharp(\eta_2) \qquad E = \bigcup_{\substack{\eta_1 \in \rho^\sharp(\ell_1)\\ \eta_2 \in \rho^\sharp(\ell_2)}} \ulcorner \eta_1 + \eta_2 \urcorner^k \qquad \sigma^{\sharp\prime} = \sigma^\sharp \bigodot_{\substack{\eta_1 \in \rho^\sharp(\ell_1),\ \eta_2 \in \rho^\sharp(\ell_2)\\ \eta = \ulcorner \eta_1 + \eta_2 \urcorner^k\\ \nu = \sigma^\sharp(\eta_1) + \sigma^\sharp(\eta_2)}} [\eta \mapsto \sigma^\sharp(\eta) \cup \nu]}{\langle \rho^\sharp, \sigma^\sharp, v_0^{\ell_0} + v_1^{\ell_1} \rangle \xrightarrow{\ell = \ell_1 + \ell_2}{}_k \langle \rho^\sharp[\ell \mapsto \rho^\sharp(\ell) \cup E], \sigma^{\sharp\prime}, v^\ell \rangle} \tag{17}$$

$$\frac{v^\sharp = \bigcup_{\substack{\eta_1 \in \rho^\sharp(\ell_1)\\ \eta_2 \in \rho^\sharp(\ell_2)}} \sigma^\sharp(\eta_1) \times^\sharp_{\mathbb{E}} \sigma^\sharp(\eta_2) \qquad E = \bigcup_{\substack{\eta_1 \in \rho^\sharp(\ell_1)\\ \eta_2 \in \rho^\sharp(\ell_2)}} \ulcorner \eta_1 \times \eta_2 \urcorner^k \qquad \sigma^{\sharp\prime} = \sigma^\sharp \bigodot_{\substack{\eta_1 \in \rho^\sharp(\ell_1),\ \eta_2 \in \rho^\sharp(\ell_2)\\ \eta = \ulcorner \eta_1 \times \eta_2 \urcorner^k\\ \nu = \sigma^\sharp(\eta_1) \times \sigma^\sharp(\eta_2)}} [\eta \mapsto \sigma^\sharp(\eta) \cup \nu]}{\langle \rho^\sharp, \sigma^\sharp, v_0^{\ell_0} \times v_1^{\ell_1} \rangle \xrightarrow{\ell = \ell_1 \times \ell_2}{}_k \langle \rho^\sharp[\ell \mapsto \rho^\sharp(\ell) \cup E], \sigma^{\sharp\prime}, v^\ell \rangle} \tag{18}$$

$$\frac{\langle \rho^\sharp, \sigma^\sharp, e_0 \rangle \xrightarrow{A}_k \langle \rho^{\sharp\prime}, \sigma^{\sharp\prime}, e_2 \rangle}{\langle \rho^\sharp, \sigma^\sharp, e_0 + e_1 \rangle \xrightarrow{A}_k \langle \rho^{\sharp\prime}, \sigma^{\sharp\prime}, e_2 + e_1 \rangle} \tag{19}$$

$$\frac{\langle \rho^\sharp, \sigma^\sharp, e_0 \rangle \xrightarrow{A}_k \langle \rho^{\sharp\prime}, \sigma^{\sharp\prime}, e_2 \rangle}{\langle \rho^\sharp, \sigma^\sharp, e_0 \times e_1 \rangle \xrightarrow{A}_k \langle \rho^{\sharp\prime}, \sigma^{\sharp\prime}, e_2 \times e_1 \rangle} \tag{20}$$

$$\frac{e \equiv_k e_1 \qquad \langle \rho^\sharp, \sigma^\sharp, e_1 \rangle \xrightarrow{A}_k \langle \rho^{\sharp\prime}, \sigma^{\sharp\prime}, e_2 \rangle \qquad e_2 \equiv_k e_3}{\langle \rho^\sharp, \sigma^\sharp, e_0 \rangle \xrightarrow{A}_k \langle \rho^{\sharp\prime}, \sigma^{\sharp\prime}, e_3 \rangle} \tag{21}$$

**Fig. 4.** The abstract semantics

This connection abstracts sets of values of $\mathbb{E}$ by intervals in a componentwise way. The partial order $\sqsubseteq^\sharp_{\mathbb{E}}$ is the componentwise inclusion order on intervals.

An expression $e$ of arbitrary height can be abstracted by $\eta \in \text{Expr}^\sharp_k$ by means of the operator $\ulcorner e \urcorner^k$ recursively defined as follows:

$$\begin{array}{rcll}
\ulcorner v^\ell \urcorner^k &=& v^\ell & k \geq 0\\
\ulcorner \top_\eta \urcorner^k &=& \top_\eta & k \geq 0\\
\ulcorner e_1 + e_2 \urcorner^0 &=& \top_\eta &\\
\ulcorner e_1 \times e_2 \urcorner^0 &=& \top_\eta &\\
\ulcorner e_1 + e_2 \urcorner^k &=& \ulcorner e_1 \urcorner^{k-1} + \ulcorner e_2 \urcorner^{k-1} & k \geq 1\\
\ulcorner e_1 \times e_2 \urcorner^k &=& \ulcorner e_1 \urcorner^{k-1} \times \ulcorner e_2 \urcorner^{k-1} & k \geq 1
\end{array}$$

Intuitively, $\ulcorner e \urcorner^k$ replaces in $e$ all the nodes of height $k$ which are not values by $\top_\eta$. The function $\ulcorner . \urcorner^k$ is indifferently applied to expressions $e \in$Expr or abstract expressions $\eta \in \text{Expr}^\sharp_{k'}$ for any integer $k'$.

The abstract semantics, given in Figure 4, uses reduction rules of the form $\langle \rho^\sharp, \sigma^\sharp, e \rangle \xrightarrow{A}_k \langle \rho^{\sharp\prime}, \sigma^{\sharp\prime}, e' \rangle$. The symbol $k$ is a parameter of the semantics and $A$ is an action indicating which operation is actually performed by the transition. Actions are used to build a new arithmetic expression from a trace and are detailed in Section 5. The environment $\rho^\sharp \ : \ \mathcal{L} \to \wp(\text{Expr}^\sharp_k)$ maps labels to sets of abstract expressions. The environment $\sigma^\sharp \ : \text{Expr}^\sharp_k \to \mathbb{E}^\sharp$ maps abstract expressions to abstract values. The symbols $\mathrm{Env}^\sharp_\rho$ and $\mathrm{Env}^\sharp_\sigma$ denote the sets of such environments. Intuitively, $\eta \in \mathrm{Expr}^\sharp_k$ abstracts a set $S$ of expressions and $\sigma^\sharp$ relates $\eta$ to an abstract value containing all the possible values resulting from the evaluation of $e \in S$.

The expression $\sigma^\sharp[\eta \mapsto v^\sharp]$ denotes the environment $\sigma^\sharp$ extended by $\sigma^\sharp(\eta) = v^\sharp$ and

$$\sigma^\sharp \bigodot_{\eta \in S,\ v^\sharp = f(\eta)} [\eta \mapsto v^\sharp]$$

is a shortcut for $\sigma^\sharp[\eta_1 \mapsto f(\eta_1)][\eta_2 \mapsto f(\eta_2)]\ldots[\eta_n \mapsto f(\eta_n)]$, for all $\eta_i$, $1 \le i \le n$, such that $\eta_i \in S$.

In Figure 4, Equation (17) and Equation (18) are used for the addition and for the product of two values, respectively. Let us assume that $v_0^{\ell_0} + v_1^{\ell_1}$ is the current expression. The values $v_1$ and $v_2$ result from the evaluation of some expressions $\eta_1 \in \rho^\sharp(\ell_1)$ and $\eta_2 \in \rho^\sharp(\ell_2)$ (assuming that, initially, $\rho^\sharp(\ell) = v^{\sharp\ell}$ for any value $v^{\sharp\ell}$ occurring in the expression.) So, $v^\sharp = \sigma^\sharp(\eta_1) + \sigma^\sharp(\eta_2)$. In Equation (17), the result $v^\sharp$ of the addition is obtained by joining the sums of all the possible operands in $\sigma^\sharp(\eta_1)$ and in $\sigma^\sharp(\eta_2)$, for all possible abstract expressions $\eta_1 \in \rho^\sharp(\ell_1)$ and $\eta_2 \in \rho^\sharp(\ell_2)$. Next, a fresh label $\ell$ is attached to $v^\sharp$ and $\rho^\sharp$ is modified by assigning to $\ell$ the set $E$ of abstract expressions which have possibly been used to compute $v^\sharp$. Finally, $\sigma^\sharp$ is updated: it is extended by assignments $[\eta \mapsto \sigma^\sharp(\eta) \cup \nu]$ where $\eta$ is one of the possible expressions used to compute $v$ and $\nu$ is the corresponding abstract value.

Equation (18) is similar to Equation (17) and equations (19) and (20) present no difficulty. Equation (21) is similar to equations (8) and (15): it introduces non-determinism in the semantics by means of a syntactic equivalence relation but now we use a new relation $\equiv_k$ instead of the previous relation $\equiv$.

**Definition 3.** *Let $\sim_k \subseteq Expr \times Expr$ be the equivalence relation defined by:*

$$e \sim_k e' \iff \ulcorner e \urcorner^k = \ulcorner e' \urcorner^k .$$

*Then $\equiv_k \subseteq Expr \times Expr$ is the quotient relation $\equiv / \sim_k$.*

Let us remark that $\equiv_k$ is coarser than $\equiv$ (which means that $\equiv \subseteq \equiv_k$) and that, while the $\equiv$-class $Cl_\equiv(e)$ of an expression $e$ contains all the expressions generated by the rules $(i)$ to $(vii)$ of Figure 2, the $\equiv_k$-class $Cl_{\equiv_k}(e)$ contains only one element of each $\sim_k$ class among the $\equiv$-equivalent elements.

At each step, our concrete and abstract semantics generate one new path per element of the equivalence class of the current expression. As $\equiv_k$ is coarser than $\equiv$, the number of paths of the semantics based on $\equiv_k$ is smaller than the number of paths of the semantics based on $\equiv$.

**Property 4.** *Let $e$ be an expression of size $n$.*

*(i) In the worst case, the $\equiv$-class of $e$ contains $O(exp(n))$ elements.*
*(ii) In the worst case, the $\equiv_k$-class of $e$ contains $O(n^k)$ elements.*

This property states that the number of expressions $\equiv_k$-equivalent to a given expression $e$ is a polynomial of degree $k$, in the size of $e$. A worst case consists of taking a sequence of sums $x_1 + x_2 + \ldots + x_n$ which, by associativity, can be evaluated in $n!$ ways using $\equiv$ and in $n^k$ ways using $\equiv_k$, where $k$ is a user-defined parameter of the semantics.

### 4.3 Correctness of the Abstract Semantics

In this section, we show that the abstract semantics of Section 4.2 is a correct abstraction of the non-standard semantics of Section 4.1. First, we relate the environments used in the non-standard semantics and in the abstract semantics by the Galois connections

$$\langle \wp(\mathrm{Env}_\rho), \subseteq \rangle \underset{\alpha_k^\rho}{\overset{\gamma_k^\rho}{\leftrightarrows}} \langle \mathrm{Env}^\sharp_{\rho,k}, \sqsubseteq_\rho \rangle \tag{22}$$

and

$$\langle \wp(\mathrm{Env}_\sigma), \subseteq \rangle \underset{\alpha_k^\sigma}{\overset{\gamma_k^\sigma}{\leftrightarrows}} \langle \mathrm{Env}^\sharp_{\sigma,k}, \sqsubseteq_\sigma \rangle. \tag{23}$$

The partial order as well as abstraction and concretization functions for the first kind of environments are defined by

$$\rho_1^\sharp \sqsubseteq_\rho \rho_2^\sharp \iff \forall \ell \in \mathrm{Dom}(\rho_1^\sharp),\ \rho_1^\sharp(\ell) \subseteq \rho_2^\sharp(\ell), \tag{24}$$

$$\alpha_k^\rho(R) = \rho^\sharp \ :\ \forall \ell \in \mathcal{L},\ \rho^\sharp(\ell) = \cup_{\rho \in R} \ulcorner \rho(\ell) \urcorner^k, \tag{25}$$

$$\gamma_k^\rho(\rho^\sharp) = \{\rho \in \mathrm{Env}_\rho \ :\ \forall \ell \in \mathcal{L},\ \ulcorner \rho(\ell) \urcorner^k \in \rho^\sharp(\ell)\}. \tag{26}$$

The environment $\rho_1^\sharp$ is smaller than $\rho_2^\sharp$ if, for any label $\ell$, the set $\rho_1^\sharp(\ell)$ is a subset of $\rho_2^\sharp(\ell)$. The abstraction $\alpha_k^\rho(R)$ of a set $R = \{\rho_1, \rho_2, \ldots, \rho_n\}$ of environments is the abstract environment $\rho^\sharp$ which maps any label $\ell$ to the set of abstract expressions $\ulcorner e \urcorner^k$ such that $\rho_i(\ell) = e$ for some $1 \le i \le n$. Conversely, $\gamma_k^\rho$ is the set of environments $\rho$ which map $\ell$ to an expression $e$ such that $\ulcorner e \urcorner^k = \rho^\sharp(\ell)$. Similarly, we have for the second kind of environments

$$\sigma_1^\sharp \sqsubseteq_\sigma \sigma_2^\sharp \iff \forall \eta \in \mathrm{Dom}(\sigma_1^\sharp),\ \sigma_1^\sharp(\eta) \sqsubseteq_{\mathbb{E}}^\sharp \sigma_2^\sharp(\eta), \tag{27}$$

$$\alpha_k^\sigma(S) = \sigma^\sharp \ :\ \forall \eta \in \mathrm{Expr}_k^\sharp,\ \sigma^\sharp(\eta) = \alpha(\{\sigma(\eta),\ \sigma \in S\}), \tag{28}$$

$$\gamma_k^\sigma(\sigma^\sharp) = \{\sigma \in \mathrm{Env}_\sigma \ :\ \forall e \in \mathrm{Expr},\ \sigma(e) \in \gamma(\sigma^\sharp(\ulcorner e \urcorner^k))\}. \tag{29}$$

The environment $\sigma_1^\sharp$ is smaller than $\sigma_2^\sharp$ if $\sigma_1^\sharp$ maps any abstract expression $\eta$ to an abstract value smaller than $\sigma_2^\sharp$. The abstraction $\alpha_k^\sigma$ and concretization $\gamma_k^\sigma$ are based on the Galois connection introduced in Section 4.2 to relate concrete and abstract values.

Let $\langle \rho, \sigma, e\rangle \longrightarrow^n \langle \rho', \sigma', v\rangle$ and $\langle \rho^\sharp, \sigma^\sharp, e\rangle \overset{A}{\longrightarrow}{}_k^n \langle \rho^{\sharp\prime}, \sigma^{\sharp\prime}, v^\sharp\rangle$ denote sequences of reduction steps of length $n$ in the non-standard and abstract semantics, yielding final values $v$ and $v^\sharp$, respectively. The following property holds.

**Property 5.** *If $\langle \rho, \sigma, e\rangle \longrightarrow^n \langle \rho', \sigma', v\rangle$ and if $\alpha_k^\rho(\rho) \sqsubseteq_\rho \rho^\sharp$ and $\alpha_k^\sigma(\sigma) \sqsubseteq_\sigma \sigma^\sharp$ then $\langle \rho^\sharp, \sigma^\sharp, e\rangle \overset{A}{\longrightarrow}{}_k^n \langle \rho^{\sharp\prime}, \sigma^{\sharp\prime}, v^\sharp\rangle$ such that $v \in \gamma(v^\sharp)$, $\alpha_k^\rho(\rho') \sqsubseteq_\rho \rho^{\sharp\prime}$ and $\alpha_k^\sigma(\sigma') \sqsubseteq_\sigma \sigma^{\sharp\prime}$.*

Property 5 states that for any path of length $n$, in the non-standard semantics which leads to a value $v$, there exists a path of the abstract semantics of length $n$ which leads to a value $v^\sharp$ such that $v \in \gamma(v^\sharp)$.

Proof

The proof is by induction on the length $n$ of the reduction sequence. If $n = 1$ then $e = v_1^{\ell_1} + v_2^{\ell_2}$ or $e = v_1^{\ell_1} \times v_2^{\ell_2}$. Let us assume that $e = v_1^{\ell_1} + v_2^{\ell_2}$ (the case $e = v_1^{\ell_1} \times v_2^{\ell_2}$ is similar). Let $v = v_1 + v_2$. In the non-standard semantics we have:

$$\langle \rho, \sigma, e\rangle \longrightarrow \langle \rho[\ell \mapsto \rho(\ell_1) + \rho(\ell_2)], \sigma[\rho(\ell_1) + \rho(\ell_2) \mapsto v], v^\ell\rangle$$

In the abstract semantics we have $\langle \rho^\sharp, \sigma^\sharp, e\rangle \overset{A}{\longrightarrow}{}_k^n \langle \rho^{\sharp\prime}, \sigma^{\sharp\prime}, v^\sharp\rangle$ with:

$$v^\sharp = \bigcup_{\substack{\eta_1 \in \rho^\sharp(\ell_1)\\ \eta_2 \in \rho^\sharp(\ell_2)}} \sigma^\sharp(\eta_1) + \sigma^\sharp(\eta_2)$$

As $\rho(\ell_1) = v_1^{\ell_1}$, $\rho(\ell_2) = v_2^{\ell_2}$, since by hypothesis, $\alpha_k^\rho(\rho) \sqsubseteq_\rho \rho^\sharp$ and $\alpha_k^\sigma(\sigma) \sqsubseteq_\sigma \sigma^\sharp$, and also because in a Galois connection, $\gamma \circ \alpha$ is extensive ($R \subseteq \gamma_k^\rho(\alpha_k^\rho(R))$ and $S \subseteq \gamma_k^\sigma(\alpha_k^\sigma(S))$), we thus have $v_1 \in \gamma(\sigma^\sharp(\rho^\sharp(\ell_1)))$ and $v_2 \in \gamma(\sigma^\sharp(\rho^\sharp(\ell_2)))$. Consequently, $v \in \gamma(v^\sharp)$.

The proof for $n = 1$ is completed without difficulty by showing that $\alpha_k^\rho(\rho') \sqsubseteq_\rho \rho^{\sharp\prime}$ and $\alpha_k^\sigma(\sigma') \sqsubseteq_\sigma \sigma^{\sharp\prime}$ with $\rho' = \rho[\ell \mapsto \rho(\ell_1)+\rho(\ell_2)]$ and $\sigma' = \sigma[\rho(\ell_1)+\rho(\ell_2) \mapsto v]$.

Now, we assume that the property holds for any $m \leq n$ and we consider a sequence of length $n + 1$. We distinguish two cases:

- Rules of Equation (13) and Equation (14): if $\frac{\langle \rho,\sigma,e_0\rangle \longrightarrow \langle \rho',\sigma',e_2\rangle}{\langle \rho,\sigma,e_0+e_1\rangle \longrightarrow \langle \rho',\sigma',e_2+e_1\rangle}$ then $\frac{\langle \rho^\sharp,\sigma^\sharp,e_0\rangle \overset{A}{\longrightarrow}_k \langle \rho^{\sharp\prime},\sigma^{\sharp\prime},e_2\rangle}{\langle \rho^\sharp,\sigma^\sharp,e_0+e_1\rangle \overset{A}{\longrightarrow}_k \langle \rho^{\sharp\prime},\sigma^{\sharp\prime},e_2+e_1\rangle}$. By our induction hypothesis, $\alpha_k^\rho(\rho') \sqsubseteq_\rho \rho^{\sharp\prime}$ and $\alpha_k^\sigma(\sigma') \sqsubseteq_\sigma \sigma^{\sharp\prime}$. Now, $\langle \rho', \sigma', e'\rangle \longrightarrow^n \langle \rho'', \sigma'', v\rangle$ and we may apply again our induction hypothesis.
- Rule of Equation (15): let us assume that $\frac{e \equiv e_1 \qquad \langle \rho,\sigma,e_1\rangle \longrightarrow \langle \rho',\sigma',e_2\rangle \qquad e_2 \equiv e_3}{\langle \rho,\sigma,e_0\rangle \longrightarrow \langle \rho',\sigma',e_3\rangle}$. Then, since, by definition of $\equiv_k$, $\equiv \subseteq \equiv_k$, $e \equiv e_1 \Rightarrow e \equiv_k e_1$ and $e_2 \equiv e_3 \Rightarrow e_2 \equiv_k e_3$. So, in the abstract semantics we have:

$$\frac{e \equiv_k e_1 \qquad \langle \rho^\sharp, \sigma^\sharp, e_1\rangle \overset{A}{\longrightarrow}_k \langle \rho^{\sharp\prime}, \sigma^{\sharp\prime}, e_2\rangle \qquad e_2 \equiv_k e_3}{\langle \rho^\sharp, \sigma^\sharp, e_0\rangle \overset{A}{\longrightarrow}_k \langle \rho^{\sharp\prime}, \sigma^{\sharp\prime}, e_3\rangle}$$

  Then we can complete the proof, by induction, in the same way as in the previous case. □

# 5 Semantics Transformation

The concrete semantics of an arithmetic expression is the floating-point semantics $\rightarrow_{\mathbb{F}}$ defined in Section 3.1. Indeed, this is the only semantics which indicates how an expression is actually evaluated by a computer. Given an expression $e$ and its (unique) execution trace $t = e \rightarrow_{\mathbb{F}}^{*} v$, the semantics transformation has to generate a new trace $t' = e' \rightarrow_{\mathbb{F}}^{*} v'$ such that $t$ and $t'$ are equal at some observational level. This is performed in Section 5.1 by using the information provided by the abstract semantics $\xrightarrow{A}_k$. In Section 5.2, we prove that $e \rightarrow_{\mathbb{R}}^{*} v''$ and $e' \rightarrow_{\mathbb{R}}^{*} v''$ for the same value $v''$, where $\rightarrow_{\mathbb{R}}$ is the semantics introduced in Section 3.2.

## 5.1 Semantics Transformation

Because the abstract semantics $\xrightarrow{A}_k$ of an expression $e$, as defined in Section 4.2, is non-deterministic, the abstract interpretation of $e$ consists of a set of traces. The semantics transformation $\tau_k$ is based on the trace $e \xrightarrow{A}{}_k^{*} v^\sharp$ which optimizes the quality of the evaluation: recall, from Section 3.3, that, in the global error based semantics, any value is a pair $(x, \mu) \in \mathbb{E}$ where $x$ is a computer representable value and $\mu$ a measure of the quality of $x$. Recall also that $(x_1, \mu_1) \prec (x_2, \mu_2) \iff \mu_1 \leq \mu_2$. Let $\mu_1^\sharp = [\underline{\mu_1}, \overline{\mu_1}]$ and $\mu_2^\sharp = [\underline{\mu_2}, \overline{\mu_2}]$. The corresponding order in $\mathbb{E}^\sharp$ is:

$$(x_1^\sharp, \mu_1^\sharp) \prec^\sharp (x_2^\sharp, \mu_2^\sharp) \iff \max(|\underline{\mu_1}|, |\overline{\mu_1}|) \leq \max(|\underline{\mu_2}|, |\overline{\mu_2}|) \qquad (30)$$

In $\prec^\sharp$, $v_1^\sharp$ is more precise than $v_2^\sharp$ if, in absolute value, the maximal error on $v_1^\sharp$ is less than the maximal error on $v_2^\sharp$.

The transformation $\tau_k$ is based on the minimal abstract trace $e \xrightarrow{A}{}_k^{*} v^\sharp$, i.e. the trace which yields the minimal value $v^\sharp$, in the sense of $\prec^\sharp$. Remark that, since $\xrightarrow{A}_k$ uses abstract values of $\mathbb{E}^\sharp$, the transformation $\tau_k$ minimizes the worst error $\mu$ which may occurs during an evaluation. Therefore, $\tau_k$ minimizes the precision lost which may arise during an evaluation in the worst case, that is for the most pessimistic combination of data.

Because the semantics $\xrightarrow{A}_k$ allows more steps than the semantics $\rightarrow_{\mathbb{F}}$ (in $\rightarrow_{\mathbb{F}}$ an expression may not be transformed by $\equiv_k$), we cannot directly transform a trace of $\xrightarrow{A}_k$ into a trace of $\rightarrow_{\mathbb{F}}$: we first have to rebuild the totally parsed expression which has actually been evaluated by $\xrightarrow{A}_k$. This is achieved by using the actions $A$ appearing in the transitions of the abstract semantics and which collect the operations actually performed along a trace.

Actions are expressions of the form $\ell = \ell_1 + \ell_2$ or $\ell = \ell_1 \times \ell_2$, where $\ell$, $\ell_1$ and $\ell_2$ are labels belonging to $\mathcal{L}$. An action $\ell = \ell_1 + \ell_2$ indicates that the value of label $\ell$ is the addition of the expressions of labels $\ell_1$ and $\ell_2$.

The expression generation function $\mathsf{P}$ is defined in Figure 5. $\mathsf{P}$ takes a trace, an environment $\iota \ : \ \mathcal{L} \rightarrow$Expr and computes a new environment $\iota'$. For a trace

$$\mathbf{P}\left(\langle\rho^\sharp,\sigma^\sharp,e\rangle \xrightarrow{\ell=\ell_1+\ell_2}{}_k \langle\rho^{\sharp\prime},\sigma^{\sharp\prime},e'\rangle,\iota\right) = \iota[\ell \mapsto \iota(\ell_1)+\iota(\ell_2)] \tag{31}$$

$$\mathbf{P}\left(\langle\rho^\sharp,\sigma^\sharp,e\rangle \xrightarrow{\ell=\ell_1\times\ell_2}{}_k \langle\rho^{\sharp\prime},\sigma^{\sharp\prime},e'\rangle,\iota\right) = \iota[\ell \mapsto \ell_1 \times \ell_2] \tag{32}$$

$$\mathbf{P}\left(\langle\rho^\sharp,\sigma^\sharp,v^{\sharp\ell}\rangle,\iota\right) = \iota(\ell) \tag{33}$$

$$\mathbf{P}\left(s_1 \xrightarrow{A}_k s_2 \xrightarrow{A}_k \ldots s_n,\iota\right) = \mathbf{P}\left(s_2 \xrightarrow{A}_k \ldots s_n, \mathbf{P}(s_1 \xrightarrow{A}_k s_2)\right) \tag{34}$$

**Fig. 5.** Generation of the new expression

$t^\sharp = \langle\rho^\sharp,s^\sharp,e\rangle \xrightarrow{A}_k^* \langle\rho^{\sharp\prime},\sigma^{\sharp\prime},v^\sharp\rangle$, initially assuming that $\iota(\ell)=v$ for any value $v^\ell$ occurring in the source expression $e$, $\mathbf{P}(t^\sharp,\iota)=\iota'(\ell)$, where $\iota'(\ell)$ is the expression actually evaluated by $t^\sharp$.

Let $e$ be an arithmetic expression and let $\mathcal{T}_k^\sharp(e)$ denote the set of evaluation traces in the abstract semantics $\xrightarrow{A}_k$ of $e$, i.e. $\mathcal{T}_k^\sharp(e) = \{\langle\rho^\sharp,\sigma^\sharp,e\rangle \xrightarrow{A}_k^* \langle\rho^{\sharp\prime},\sigma^{\sharp\prime},v^\sharp\rangle\}$. The minimal trace of $\mathcal{T}_k^\sharp(e)$ is

$$\min_{\prec^\sharp}\mathcal{T}_k^\sharp(e) = \langle\rho^\sharp,\sigma^\sharp,e\rangle \xrightarrow{A}_k^* \langle\rho^{\sharp\prime},\sigma^{\sharp\prime},v^\sharp\rangle,$$

where $v^\sharp \prec^\sharp v^{\sharp\prime}$ whenever $\langle\rho^\sharp,\sigma^\sharp,e\rangle \xrightarrow{A}_k^* \langle\rho^{\sharp\prime},\sigma^{\sharp\prime},v^{\sharp\prime}\rangle \in \mathcal{T}_k^\sharp(e)$.

The transformation is defined as follows:

**Definition 6.** *Let $e$ be an arithmetic expression. The semantics transformation $\tau_k$ of $e \rightarrow_{\mathbb{F}} v$ is defined by*

$$\tau_k\left(e \rightarrow_{\mathbb{F}} v, \mathcal{T}_k^\sharp(e)\right) = \mathbf{P}\left(min_{\prec^\sharp}\mathcal{T}_k^\sharp(e)\right) \rightarrow_{\mathbb{F}} v'. \tag{35}$$

By Definition 6, the transformed trace is the evaluation trace in the floating-point arithmetic based semantics of the expression $\mathbf{P}(e)$ generated from the minimal trace $e \xrightarrow{A}_k^* v^\sharp = \min_{\prec^\sharp}\mathcal{T}_k^\sharp(e)$.

### 5.2 Correctness of the Transformation

In order to prove the correctness of the transformation, we show that, at an observational level [6], the semantics of the original expression $e$ and the semantics of the transformed expression $e_t$ are equal. Our observation consists of showing that $e$ and $e_t$ compute the same thing in the exact arithmetic of real numbers.

Let $\alpha_{\mathcal{O}}$ be an observational abstraction $\alpha_{\mathcal{O}} : \mathbb{E} \rightarrow \mathbb{R}$ which transforms a floating-point number with errors into a real number, i.e. $\alpha_{\mathcal{O}}(x,\mu) = x+\mu$. We first introduce a lemma concerning the non-standard semantics.

**Lemma 7.** *Let $e$ be an arithmetic expression and let $\langle\rho,\sigma,e\rangle \longrightarrow^* \langle\rho',\sigma',v_1\rangle$ and $\langle\rho,\sigma,e\rangle \longrightarrow^* \langle\rho'',\sigma'',v_2\rangle$ be two paths of the non-standard semantics. Then $\alpha_{\mathcal{O}}(v_1) = \alpha_{\mathcal{O}}(v_2)$.*

Lemma 7 stems from the fact that, in $\mathbb{E}$, the errors are exactly computed. So, from the perspective of $\alpha_{\mathcal{O}}$, the traces of $\rightarrow_{\mathbb{E}}$ are identical to the traces of $\rightarrow_{\mathbb{R}}$.

**Lemma 8.** *Let $e_t = \mathsf{P}\left(t^\sharp, \iota\right)$ for some trace $\langle \rho^\sharp, s^\sharp, e\rangle \xrightarrow{A}{}_k^{*} \langle \rho^{\sharp\prime}, \sigma^{\sharp\prime}, v^\sharp\rangle \in \mathcal{T}_k^\sharp(e)$. Then $e \equiv e_t$.*

As a consequence, in the non-standard semantics $e$ and $e_t$ lead to observationally equivalent values. By Lemma 7, if $\langle \rho, \sigma, e\rangle \longrightarrow^* \langle \rho', \sigma', v\rangle$ then, by the rule of Equation (15), $\langle \rho, \sigma, e_t\rangle \longrightarrow^* \langle \rho'', \sigma'', v\rangle$. Using Lemma 7 and Lemma 8, we have:

**Property 9.** *Let $e$ be an arithmetic expression, let $t = e \rightarrow_{\mathbb{F}}^* v$ be the concrete evaluation trace of $e$, and let $t^\sharp = \langle \rho^\sharp, s^\sharp, e\rangle \xrightarrow{A}{}_k^{*} \langle \rho^{\sharp\prime}, \sigma^{\sharp\prime}, v^\sharp\rangle \in \mathcal{T}_k^\sharp(e)$. We have:*

$$e \rightarrow_{\mathbb{R}}^* v_{\mathbb{R}} \iff \mathsf{P}(t^\sharp, \iota) \rightarrow_{\mathbb{R}}^* v_{\mathbb{R}} \tag{36}$$

In particular, this property holds for the minimal trace used in Equation (35), in the definition of $\tau_k$.

## 6 Experimental Results

A prototype based on the abstract semantics of Section 4.2 and on the transformation of Section 5 has been implemented and, in this section, we present some experimental results.

As explained in Section 2, adding numbers of different magnitudes may lead to important precision loss by absorption. For example, in the IEEE754 simple-precision format, $1.0 + 5e^{-8} = 1.0$ while $1.0 + (2 \times 5e^{-8}) \neq 1.0$. We consider the expression:

$$e = a \times ((b + c) + d)$$

and the global abstract environment $\theta^\sharp$ such that:

$$a = [56789, 98765] \quad b = [0, 1] \quad c = [0, 5e^{-8}] \quad d = [0, 5e^{-8}] \tag{37}$$

Our prototype computes for this example (with $k = 2$):

```
(a*((b+c)+d)) -> ((a*b)+(a*(c+d)))
```

The sums are parsed in order to first add the smallest terms: this limits the absorption. Furthermore, the product is distributed and this avoids the multiplication of roundoff errors of the additions by a large value and, consequently, this also reduces the final error.

Using the domain $\mathbb{E}^\sharp$, which computes an over-approximation of the error attached to the result of a floating-point computation, our prototype also outputs a bound on the maximal error arising during the evaluation of an expression (for any concrete set of inputs in the intervals given in Equation (37)). The errors for the source and transformed expressions are:

- Error bound on `(a*((b+c)+d))`: [-1.5679E-2,1.5680E-2]
- Error bound on `((a*b)+(a*(c+d)))`: [-7.8125E-3,7.8126E-3]

The error on the transformed expression is approximatively half the error on the original expression.

Our second example concerns the sum $s = \sum_{i=0}^{4} x_i$, with $x_i = [2^i, 2^{i+1}]$. The results, for different values of $k$ are given in the table below, where a, b, c, d and e stand for $x_0$, $x_1$, $x_2$, $x_3$ and $x_4$, respectively:

| Case | Expression | Error bound |
|---|---|---|
| Source expression | (((e+d)+c)+b)+a | [-7.6293E-6,7.6294E-6] |
| $k = 1$ | (b+a)+(c+(e+d)) | [-5.9604E-6,5.9605E-6] |
| $k = 2$ | (c+(b+a))+(e+d) | [-4.5299E-6,4.5300E-6] |
| $k = 3$ | (d+(c+(a+b)))+e | [-3.5762E-6,3.5763E-6] |

As the parameter $k$ increases, the terms are more and more sorted, increasing the precision of the result. With $k = 3$, the error is guaranteed to be less than half the error on the original expression.

Another class of examples concerns the evaluation of polynomials. Again, anybody familiar with computer arithmetic knows that, in general, factorization improves the quality of the evaluation of a polynomial. In the abstract environment $\theta^\sharp$ in which an initial error has been attached to $x$: for $x = ([0, 2], [0, 0.0005])$, we obtain the following results:

| Case | Expression | Error bound |
|---|---|---|
| Source expression | x+(x*x) | [-1.800074334E-3,1.001074437E-3] |
| $k = 2$ | (1.0+x)*x | [-9.000069921E-4,1.010078437E-4] |
| Source expression | (x*(x*x))+(x*x) | [-1.802887642E-3,3.191200091E-3] |
| $k = 3$ | (x+1.0)*(x*x) | [-1.818142851E-4,1.390014781E-3] |
| $k = 4$ | ((1.0+x)*x)*x | [-9.091078216E-5,1.100112212E-3] |

Our last example concerns the expression $(a + b)^2$. If $b \ll a$, then we obtain a better precision by developing the remarkable identity. Using $a = [5, 10]$ and $b = [0, 0.001]$, our prototype outputs the following results.

| Case | Expression | Error bound |
|---|---|---|
| Source expression | (a+b)*(a+b) | [-1.335239380E-5,1.335239381E-5] |
| $k = 2$ | ((b*(a+b))+(a*b))+(a*a) | [-7.631734013E-6,7.631734014E-6] |
| $k = 3$ | (((b*a)+(b*b))+(b*a))+(a*a) | [-7.631722894E-6,7.631722895E-6] |

With $k = 3$ the transformation consists of finding the remarkable identity. However, with $k = 2$, another formula which significantly improve the precision has already been found.

## 7 Perspectives

We believe that the new kind of program transformation introduced in this article can be improved and extended in many ways.

First of all, we aim at extending our methodology to full programming languages, with variables, loops and conditionals, instead of simple arithmetic expressions. We believe it is possible to rewrite computations defined among many lines of code. General code transformation techniques [10] could be used. For example, loop unfolding techniques can be used to improve the numerical precision of iterative computations. In addition, some statements may also introduce precision loss, like assignments when processor registers have more digits than memory locations [11]. This last remark also makes us believe that our program transformation could be used on assembler codes, possibly at compile-time. We are confident in the feasibility of such transformations for large scale programs, static analyses for numerical precision having already been defined for general programming languages and being implemented in analyzers used in industrial contexts [8].

Another research direction concerns the abstract semantics. In this article, we have presented a simple abstract semantics which could be improved in many ways. For arithmetic expressions, more subtle abstractions could be defined, which more globally minimize the error on an evaluation path. The semantics of error series [13] could be useful in this context, but we believe that other approaches could also be successfully developed.

The relation $\equiv$, introduced in Section 2, identifies expressions which are equal in the reals. These laws enable us to rewrite expressions. However, the relation $\equiv$ is not unique and could be extended by many other laws. For example, some laws can be used to improve the precision of floating-point computations, like Sterbenz's theorem for subtraction [14]. Other laws can be found in [3,4,2].

Finally, other applications could be studied. For example, finite precision arithmetic is widely used in embedded systems. In order to implement a chain of operations, the programmer often works as follows: the size of the inputs (their number of digits) is know, and the result $r$ of each elementary operation is stored in a new number large enough to represent exactly $r$. Obviously, the designer of an embedded system aims at limiting the sizes of the numbers and this strongly depends on how the formula is implemented. Yet other applications, like code obfuscation for arithmetic expressions without loss of precision, could also be developed, the framework of semantics program transformation having already been used in this context [7].

## 8 Conclusion

In this article, we have introduced a semantics-based program transformation for arithmetic expressions, in order to improve the quality of their implementation. This work is a first step towards the automatic improvement of large scale codes containing numerical computations. This research direction could find many applications, in the context of embedded softwares as well as for numerical codes. In addition, this program transformation can be used either as a source to source transformation or at compile-time, during the low-level code generation phase.

We believe that our method can be improved and extended in many directions and some issues have been discussed in Section 7. Meanwhile, the experimental results of Section 6 show that the transformation of simple arithmetic expressions, using a simple analysis, already yield interesting results.

We also believe that the framework of semantics program transformation [6] was very helpful to define our method, which would have been more difficult to design and prove at the syntactic level.

More generally, our approach relies on the assumption that, concerning numerical precision, a program can be viewed either as a model or as an implementation. More precisely, a formula occurring in a source code may be considered as the specification of what should be computed in the reals as well as a sequence of machine operations. We used the first point of view to generate a new sequence of operations. We believe that this approach may lead to many further developments in the domain of program transformation for numerical precision, independently of the techniques used in this article which represent our first attempt to automatically improve the accuracy of numerical programs.

## References

1. ANSI/IEEE. IEEE Standard for Binary Floating-point Arithmetic, std 754-1985 edition (1985)
2. Bohlender, G., Walter, W., Kornerup, P., Matula, D.W.: Semantics for exact floating-point operations. In: Symp. on Computer Arithmetic, pp. 22–26 (1991)
3. Boldo, S., Daumas, M.: Properties of the subtraction valid for any floating point system. In: 7th International Workshop on Formal Methods for Industrial Critical Systems, pp. 137–149 (2002)
4. Boldo, S., Daumas, M.: Representable correcting terms for possibly underflowing floating point operations. In: Bajard, J.-C., Schulte, M. (eds.) 16th Symposium on Computer Arithmetic, pp. 79–86 (2003)
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points. In: Principles of Programming Languages 4, pp. 238–252. ACM Press, New York (1977)
6. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation. In: Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, pp. 178–190. ACM Press, New York (2002)
7. Dalla Preda, M., Giacobazzi, R.: Control code obfuscation by abstract interpretation. In: International Conference on Software Engineering and Formal Methods, SEFM'05, pp. 301–310. IEEE Computer Society Press, Los Alamitos (2005)
8. Goubault, E., Martel, M., Putot, S.: Some future challenges in the validation of control systems. In: ERTS'06 (2006)
9. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, Springer, Heidelberg (2006)
10. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Int. Series in Computer Science. Prentice Hall International, Englewood Cliffs (1993)
11. Martel, M.: Validation of assembler programs for DSPs: a static analyzer. In: Program analysis for software tools and engineering, pp. 8–13. ACM Press, New York (2004)
12. Martel, M.: An overview of semantics for the validation of numerical programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 59–77. Springer, Heidelberg (2005)
13. Martel, M.: Semantics of roundoff error propagation in finite precision calculations. Journal of Higher Order and Symbolic Computation 19, 7–30 (2006)
14. Sterbenz, P.H.: Floating-point Computation. Prentice-Hall, Englewood Cliffs (1974)

# A Fast Implementation of the Octagon Abstract Domain on Graphics Hardware

Francesco Banterle and Roberto Giacobazzi

Dipartimento di Informatica
Università degli Studi di Verona
Strada Le Grazie 15, 37134 Verona, Italy
frabante@gmail.com, roberto.giacobazzi@univr.it

**Abstract.** We propose an efficient implementation of the Octagon Abstract Domain (OAD) on Graphics Processing Unit (GPU) by exploiting stream processing to speed-up OAD computations. OAD is a relational numerical abstract domain which approximates invariants as conjunctions of constraints of the form $\pm x \pm y <= c$, where $x$ and $y$ are program variables and $c$ is a constant which can be an integer, rational or real. Since OAD computations are based on matrices, and basic matrix operators, they can be mapped easily on Graphics Hardware using texture and pixel shader in the form of a kernel that implements matrix operators. The main advantage of our implementation is that we can achieve sensible speed up by using a single GPU, for each OAD operation. This can be the basis for an efficient abstract program analyzer based on a mixed CPU-GPU architecture.

**Keywords:** Octagon Abstract Domain, General Processing on GPU, Parallel Computing, Abstract Interpretation, Static Program Analysis.

## 1 Introduction

The study of stream processing (computing and programming) is recently gaining interest as an alternative and efficient methodology for allowing parallel processing in many fields of computer science. The paradigm is essentially based on defining a set of compute-intensive operations (called kernels) which are applied to each element in the stream. The growing success of this technology is related with the impressive grow in computational power of dedicated stream processing units (e.g., for graphic processing and more in general for digital signal processing) and in their relatively cheap costs. Recently researchers have become interested in developing algorithms for GPUs. These algorithms were at the beginning designed only for Computer Graphics purposes, but the high computational power offered pushed researchers to explore the possibilities of using GPU in more general tasks, leading to the so called General Purpose Computing on GPU (GP-GPU). GPUs were applied with success in various fields: Database [10], numerical methods [2,14,9], and scientific computing [11,8]; see [21,23] for excellent surveys on GP-GPU.

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 315–332, 2007.

In this paper we propose a new programming methodology to handle massive computations in numerical (relational) abstract domains. We consider stream processors and programming as an efficient and fast methodology for implementing the basic operators of abstract domains involving large matrices and massive data sets. Among the wide spectrum of numerical abstract domains, octagons [18] plays a key role due to their relational structure and affordable computational costs. Octagons provide a way to represent a system of simplified inequalities on the sum and difference of variable pairs, i.e. they represent constraints of the form $\pm x \pm y <= c$, where $x$ and $y$ are program variables and $c$ is a constant which can be an integer, rational or a real number automatically inferred. Their typical implementation is based on Difference Bound Matrices (DBM), a data structure used to represent constraints on differences of pairs of variables. Efficiency is a key aspect in this implementation: The space is constrained for $n$-variables in $O(n^2)$ and time is constrained in up to $O(n^3)$. This makes the relational analysis based on octagons applicable to large scale programs, e.g., those considered in the Astrée static analyzer [5] which employs programs having more than 10.000 global variables, most of them floating-point, in long-time iterations (about $3.6 \times 10^6$ iterations of a single loop). We prove that an important speed-up factor can be obtained by handling DBM in a stream-like computation model. In particular we exploit the structure of Graphics Processing Unit or (GPU), also called Visual Processing Unit, for an efficient and fast implementation of the abstract domain of octagons, in particular for a fast implementation of the basic operations on DBM. GPUs provide a dedicated hardware architecture for graphics rendering by exploiting a highly parallel structure making graphic computations far more effective than typical CPUs for a wide range of complex algorithms. This architecture is particularly suitable for operations on matrices, and therefore for handling operations on DBM. A typical GPU implements a number of graphics primitive operations in a way that implements stream processing: First, the data is gathered into a stream from memory. The data is then operated upon by one or more kernels, where each kernel comprises several operations. Finally, the live data is scattered back to memory. The static analyzer designed in our implementation is based on a CPU which manages the control flow and the GPU which performs the basic operators on the domain. Octagons are represented as 2D textures and the basic operations on octagons are implemented as kernel operations on their fragments. These operations can be performed in parallel on the texture due to their independence and thanks to the high degree of parallelism of the SIMD architecture like GPUs. The pipeline of the analyzer is therefore as follows: each time the analyser reaches a program point, the corresponding instruction is decomposed into basic abstract operations on octagons. The GPU is then activated to perform the basic computations, leaving the result in the video memory. As a result of our implementation, we obtain a sensible speed-up of several orders for simple operators on octagons (i.e., intersection, union, assignment, test guard, widening), and a speed-up around 24 times for the basic operation of octagon closure, which is performed each time octagons have to be merged. The bottleneck in our system is given by the test guard operator. This is due to the SIMD architecture of GPU which requires the spreading of the computation along the whole texture.

## 2 The Octagon Abstract Domain

The octagon abstract domain introduced by Miné in [18] is a (weakly) relational abstract domain which provides an upper approximation of program invariants as conjunctions of constraints of the form $\pm x \pm y <= c$, where $x$ and $y$ are program variables and $c$ is a constant which can be an integer, rational or a real. This abstract domain fits between the less precise linear-time non-relational abstract domain of intervals and the exponential-time relational approximation of convex polyhedra. Given a set of program variables $\mathcal{V} = \{V_1, \ldots, V_n\}$, we consider a set of enhanced variables: $\mathcal{V}' = \{V'_1, \ldots, V'_{2n}\}$ where for any $V_i \in \mathcal{V}$ we have both a positive form $V'_{2i-1}$, denoted $V_i^+$, and a negative form $V'_{2i}$, denoted $V_i^-$, in $\mathcal{V}'$. A Difference Bound Matrix, or DBM for short, $\mathbf{m}$ is a $n \times n$ square matrix with elements in a field $\mathbb{Z}$ or $\mathbb{R}$ [18]. The element at line $i$ and column $j$, denoted $\mathbf{m}_{ij}$ equals a constant $c$ if there is a constraint of the form $V_j - V_i \leq c$, and $+\infty$ otherwise. Thus a conjunction of octagonal constraints in $\mathcal{V}$ can be represented as a DBM with $2n$ dimension. In particular, a Galois connection has been established between DBM and sets of tuples of values:

$$\gamma(\mathbf{m}) = \{\, \langle v_1, \ldots, v_{2n} \rangle \,|\, \forall i, j \leq 2n.\ v_j - v_i \leq \mathbf{m}_{ij} \,\} \cap \{\, \langle v_1, \ldots, v_{2n} \rangle \,|\, v_{2i-1} = -v_{2i} \,\}$$

is the octagon represented by the $2n$ dimension DBM $\mathbf{m}$. This $2n$ space is isomorphic to a $n$-dimensional space which represent a convex structure having an octagon-like shape.

The set of (coherent) DBM, denoted $\mathbf{cDBM}$, enriched with a bottom (empty) element $\perp^{\mathbf{cDBM}}$ representing the empty set $\varnothing$ and a top element $\top^{\mathbf{cDBM}}$ representing the whole space, i.e., such that $\forall i, j.\ \top^{\mathbf{cDBM}}_{ij} = +\infty$, and ordered w.r.t. set inclusion, i.e., $\mathbf{m} \sqsubseteq \mathbf{n}$ iff $\forall i, j:\ \mathbf{m}_{ij} \leq \mathbf{n}_{ij}$, forms a complete lattice, where a DBM is coherent if $\forall i, j.\ \mathbf{m}_{i,j} = \mathbf{m}_{\bar{\imath}\bar{\jmath}}$ where $\bar{\imath} =$ if $i\ mod\ 2 = 0$ then $i-1$ else $i+1$. Intuitively a cDBM does not change by switching positive with negative forms of the same variable. The switch operation $\bar{\imath}$ is typically implemented by a bit-wise `xor` operation. The other classic lattice operators are defined as follows:

$$\forall \mathbf{m}, \mathbf{n} \in \mathbf{cDBM}.\ (\mathbf{m} \sqcup^{\mathbf{cDBM}} \mathbf{n})_{ij} = \mathtt{max}(\mathbf{m}_{ij}, \mathbf{n}_{ij})$$
$$\forall \mathbf{m}, \mathbf{n} \in \mathbf{cDBM}.\ (\mathbf{m} \sqcap^{\mathbf{cDBM}} \mathbf{n})_{ij} = \mathtt{min}(\mathbf{m}_{ij}, \mathbf{n}_{ij})$$

The main result in the construction and representation of the octagon abstract domain is the existence of the best abstraction of octagons as an element in $\mathbf{cDBM}$. This is achieved by computing *normal forms* for DBM representing octagons. A modified version of the Floyd-Warshall closure algorithm which performs *strong closure* is considered for this task. The intuition is that, while the Floyd-Warshall closure algorithm can be seen as a constraint propagation which completes a set of constraints until the following closure holds:

$$\begin{cases} V'_i - V'_k \leq a \\ V'_k - V'_j \leq b \end{cases} \Longrightarrow V'_i - V'_j \leq a + b$$

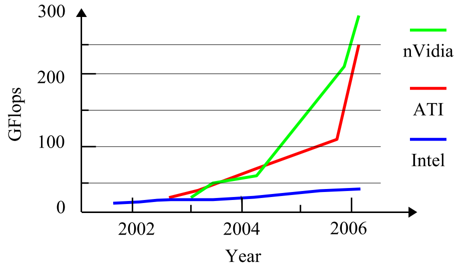


**Fig. 1.** A comparison of floating point performances between Intel CPU, ATI GPU and nVidia GPU in the last five years. As the graph shows GPU are increasing their performance every year faster than CPUs.

The modified Floyd-Warshall strong closure algorithm adds a second form of constraints until the following closure holds [18]:

$$\left\{\begin{array}{l} V'_{\bar{\imath}} - V'_i \leq a \\ V'_j - V'_{\bar{\jmath}} \leq b \end{array}\right. \Longrightarrow V'_j - V'_i \leq (a+b)/2$$

The constraints introduced by the (strong) closure algorithm are called *implicit* in order to distinguish them from the *explicit* constraints considered to build the octagon. The strong closure operation on DBM is denoted $(\cdot)^{\bullet}$. All the standard lattice-theoretic operations and C-like transfer functions have been defined on **cDBM** in order to derive an abstract semantics which has been proved correct by abstract interpretation [18].

## 3 An Overview on GPUs

In the last few years graphics hardware, known as GPU, dramatically increased its computationally power and flexibility for answering the need to increase the realism in videogames and other graphics applications. GPUs are quite a cheap product and they offer high performances, for example in the case of nVidia GeForce7950-GX2 (a double GPU equipped with 512Mb of RAM) the cost is around 310 euro (March 2007) offering a peak of 384 GFlops with a 51.2 Gb/sec bandwidth through the video memory, see figure Figure 2 for a complete sight on how a GPU is inserted in the traditional computer architecture. These performances are more than tripled compared with its predecessor the nVidia GeForce6800 Ultra, 58 GFlops with a 38.5 Gb/sec bandwidth. Indeed GPUs have an average yearly rate of growth around 2.0, which actually is higher than Moore's Law growth rate for CPU, 1.4 per year. See Figure 1 for the trend of GFlops in GPU in the last years.

### 3.1 The Programmable Graphics Pipeline

GPUs are optimized to render a stream of geometric primitives (point, lines and triangles), called *vertex buffer*, onto an array of pixels called *frame buffer*. The GPUs became in the last years fully programmable for transforming and lighting vertices and pixels. The main purpose of GPU is to processes vertices and pixels. This processing follows the classic graphics pipeline, allowing in certain stages

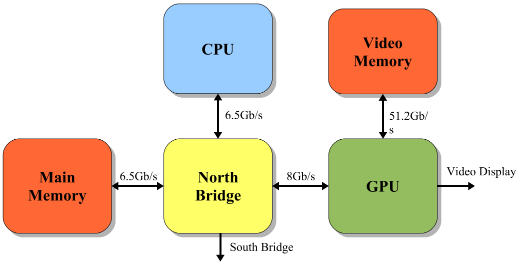


**Fig. 2.** The GPU in a traditional PC system: the GPU has a very large bandwidth with its memory (a peak of 51.2Gb/s in the nVida GeForce7900GTX). This value is nearly 8 times compared to the one for the CPU and its memory.

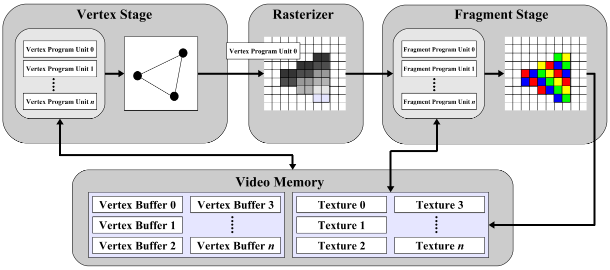


**Fig. 3.** The GPU Pipeline: vertices which define a primitive (triangle, point, line) are transformed using a function (implemented as a Vertex Program) in the Vertex Stage by Vertex Program Unit. Vertices are elaborated in parallel by many Vertex Program Units (Vertex Program Unit 0, ..., Vertex Program Unit $n$). Then primitives are discretized in fragments by the Rasterizer which passes these fragments to the Fragment Stage. At each fragment is applied a function (Fragment Program) in parallel using many Fragment Program Units (Fragment Program Unit 0, ..., Fragment Program Unit $n$). The result of the Fragment Stage is saved in a texture in the video memory.

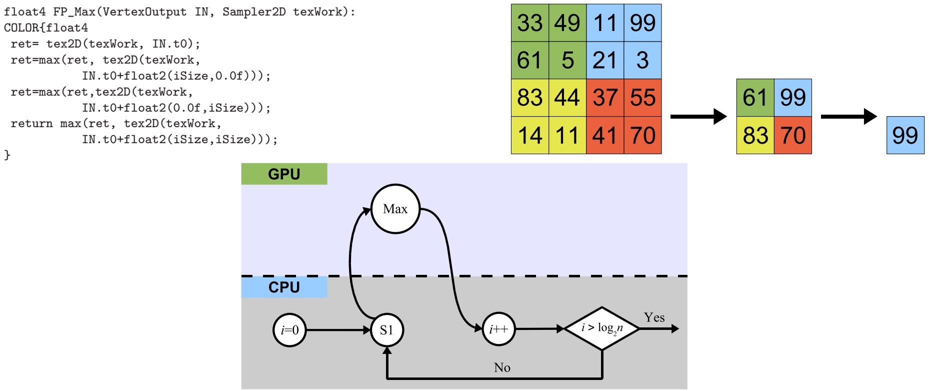
programmability via micro programs. We can see a GPU pipeline in Figure 3. The first step in the pipeline is the vertices processing. Each vertex of a vertex buffer is processed by a Vertex Program Unit (VPU) which is a programmable unit that executes a vertex program (VP). A VP is a set of instructions that specifies how a vertex will be processed by the VPU. The output of a VPU is not only a modified position for a vertex but it can also add new properties to the vertex like a color, an address for fetching the memory during the next phase etc... A modern GPU presents more than one VPU (around 6) and it automatically distributes vertices to elaborate them fast between VPUs. Note that when a GPU is processing a single vertex buffer the VP is the same for all VPUs. After the VPU, a unit called rasterizer, generates fragments of the primitives, in other words it discretizes them in pixels and it interpolates values between vertices using linear interpolation. In the last step of the pipeline the fragments created by the rasterizer are processed by another programmable unit called Fragment Program Unit (FPU), which executes a fragment program (FP). As VP, the FP is a set of instructions which specifies how a fragment will be processed by the FPU. The output, which can be a single value or a vector of values, can be stored in the frame buffer for visualization or in a texture for future processing.

As for VPUs, FPUs are numerous in a modern GPU (around 32), and fragments are automatically distributed to FPUs. Note that as for the VP, the same FP is executed by all FPUs until all fragments generated by rasterizer are completed. In this current generation of GPUs the main instructions allowed in the VPs and FPs are: float point operations (addition, subtraction, division, multiplication, square root, logarithm, etc...), random access to the video memory, assignment command, static and dynamic branching (a costly operation), and loop (with limited loop size for avoiding infinite loops). VPs and FPs are usually written in a high-level programming language similar to C. These languages are called *shading languages* because they are designed for generating images. The most common shading languages are: C for Graphics (Cg) [16], the OpenGL Shading Language (GLSL) [13] and High Level Shading Language (HLSL) [1]. These languages provide an abstraction for a very close level to the hardware, indeed they manage directly vertices, textures, fragments, etc... which are very specific graphics primitives. Other languages present an higher abstraction avoiding direct manipulation of graphics primitives and supporting GP-GPU such as SH [17], BrookGPU [3], and etc... The main disadvantage of these higher abstractions is that they are implemented on top of Graphics API such as OpenGL and Direct3D, so the overhead is quite high.

### 3.2 Kernel Programming

Data parallelism is the key for high performance in GPUs. In this section, we shortly introduce the GPU programming model called Kernel Model. The most powerful components in the GPU architecture are FPUs, because they are more numerous than VPUs, usually in a ratio 6:1, allowing more parallel power. A GP-GPU program usually uses FPUs as main processing unit. The first algorithm is segmented into independent parallel parts, called kernels, each of these is implemented in a FPU. Inputs and outputs of each kernel are arrays of data, called texture, and they are stored in the video memory. A texture can be indexed in 1D (1D texture), in 2D (2D texture), and in 3D (3D texture). Note that 1D texture can have a size of only 4096 values, while 2D texture and 3D texture can allow a size up to respectively $4096^2$ and $512^3$ values. These are the following steps for kernel to run a kernel:

1. Vertices are passed to the GPU, in order to feed the vertex stage. A typical GP-GPU invocation is a quadrilateral, parallel to the screen of the display, which covers a region of pixels that match precisely with the desired size of the output texture. In our case this provides the extreme boundaries (a quadrilateral) of a temporary address space for allocating DBMs.
2. The rasterizer generates a fragment for every pixel in the quadrilateral. In our case the rasterizer fills the adress space with all the addresses, called fragments.
3. Each fragment is processed by the FPU. At this stage all the FPUs are processing the same fragment program. The fragment program can arbitrarily read from textures in the video memory, but can only write to memory corresponding to the location of the fragment in the frame buffer determined



```
float4 FP_Max(VertexOutput IN, Sampler2D texWork):
COLOR{float4
 ret= tex2D(texWork, IN.t0);
 ret=max(ret, tex2D(texWork,
          IN.t0+float2(iSize,0.0f)));
 ret=max(ret,tex2D(texWork,
          IN.t0+float2(0.0f,iSize)));
 return max(ret, tex2D(texWork,
          IN.t0+float2(iSize,iSize)));
}
```

**Fig. 4.** The fragment program for calculating the maximum in a texture by reduction: IN is the input value of the FP calculated by interpolation from the rasterizer (IN.t0 is the vector that stores the coordinates of the current fragment), texWork is a 2D texture (declared as Sampler2D) in which are stored the values to reduce. The iSize is a constant value set in the FP, and represents the inverse of the size of the texture. tex2D is required to access to a texture in the video memory. Below the control flow of the calculation of maximum value using reduction paradigm. First the CPU sets the counter $i = 0$, then it enters in a loop. In the loop CPU transfers the control to GPU which calculates the maximum of every square of four pixels. After the GPU finishes it releases the control to CPU which increments $i$ and it tests $i > log_2(n)$ where $n$ is the width of the texture. If the guard is true the reduction is completed otherwise the CPU returns in S1.

by the rasterizer. The domain of the computation is specified for each input texture by specifying texture coordinates at each of the input vertices of the quadrilateral. In our case, the fragment program computes locally to a single pixel the operations in the octagon domain.

4. The output of the fragment program is a value, that is stored in a texture.

An algorithm needs to reiterate this process as many times as required by the computation. This is called *multi-pass*.

### 3.3 Reduction

A constraint of current GPU's generation is that a FPU cannot randomly write the result of its job in the video memory, but only on an address which is chosen by the GPU's rasterizer. This is a problem if we want to compute properties from a texture such as maximum, or minimum value. The solution is a process called Reduction: the kernel program gets as input the value of four neighbor pixels and computes the needed function using these four values. At the end it saves the output in a texture which is reduced by one half. This process is iterated until the texture is only one single pixel; see Figure 4 for a visual example of the mechanism of reduction.

### 3.4 Addresses Issues in GPUs

In the current GPUs, integer arithmetic is not supported, only floating point is allowed. This feature will be present only in the upcoming GPU generation, R600 and GeForce Series 8 recently available. In particular when we want to access to a texture we need to use float addresses. The operator $\cdot^{-}$ is used in the octagon domain to access to $v_j^{-} = -v_j$ the negative variable. This operator is defined as $i^{-} \overset{\text{def}}{=} i \texttt{ xor } 1$. As we said before the XOR operation cannot be directly performed in the fragment. A solution to this problem is to encode in a single 1D texture, called XORTexture, all the XOR values for an address and save them in float values. Therefore, every time that we need to calculate a `xor` value of an address, we fetch the XORTexture with the address.

### 3.5 Float Precision Issues

GPUs can use two different types of floating point arithmetics: single precision, close to the 32-bit IEEE-754 standard, and half precision a 16-bit (10bits mantissa, 5bits exponent and 1bit sign). The main advantage of the half precision format is that a GPU performs nearly two times the speed of single precision, however they are not precise enough for numerical application, such as those employed in the octagon domain. We observed some numerical instabilities in the closure operation, therefore we decided to use the single precision format. This format has enough precision so there is no need to implement double precision in emulation using two single precision values (value and residual) [7]. GPUs present some issues with floating point arithmetic, because they do not implement the full IEEE-754 standard [12]. While GPUs can perform precise arithmetic operations (add, subtraction, multiplication, division, and tests) they cannot handle NaN value, and partially ±Inf. Also isnan and isinf functions are very dependent by the drivers of the vendors which usually strongly suggest to avoid them in fragment programs. This can be a problem in order to represent $\top^{\mathbf{cDBM}} = +\infty$ value in the octagon domain, however we simply solved this problem by assigning to $\top^{\mathbf{cDBM}}$ the value $3.4e38$, the maximum value representable in the GeForce series 6 and 7 architecture [22].

## 4 Mapping Octagon Abstract Domain on GPU

The Octagon Abstract Domain can be naturally mapped to GPUs because the data structure used to represent octagons, the DBM matrix, can be mapped one to one with a 2D texture of GPU. So there is no need to develop a particular data structure as in [15]. Another advantage is that the operators of the OAD are very simple matrix operators that can be performed by using simple kernel programming. In our implementation we did not use any API for GP-GPU. To avoid overheads, we directly wrote the code by using OpenGL and Cg language for writing fragment programs.

### 4.1 Closure

In the octagon domain closure represents the normal form for the DBM, which calculates all the constraints (implicit and explicit) between the variables. This operation is defined in [18] by the following algorithm, which is a modified version of the Floyd-Warshall shortest-path algorithm:

$$\begin{cases} \mathbf{m}_0 \overset{\text{def}}{=} \mathbf{m} \\ \mathbf{m}_{k+1} \overset{\text{def}}{=} S(C_{2k}(\mathbf{m}_k)) \qquad \forall k\colon 0 \leq k \leq n \\ (\mathbf{m})^{\bullet} \overset{\text{def}}{=} \mathbf{m}_n \end{cases} \tag{1}$$

where $\mathbf{m}$ is an empty octagon, $C$ is defined as

$$[C_k(\mathbf{n})]_{ij} \overset{\text{def}}{=} \begin{cases} 0 & \text{for } i = j \\ \min\big(\mathbf{n}_{ij}, (\mathbf{n}_{ik} + \mathbf{n}_{kj}), & \\ \qquad (\mathbf{n}_{i\bar{k}} + \mathbf{n}_{\bar{k}j}), & \text{elsewhere} \\ \quad (\mathbf{n}_{ik} + \mathbf{n}_{k\bar{k}} + \mathbf{n}_{\bar{k}j}), & \\ \quad (\mathbf{n}_{i\bar{k}} + \mathbf{n}_{\bar{k}k} + \mathbf{n}_{kj})\big) & \end{cases} \tag{2}$$

and $S$ as

$$[S(\mathbf{n})]_{ij} \overset{\text{def}}{=} \min(\mathbf{n}_{ij}, (\mathbf{n}_{i\bar{i}}\mathbf{n}_{\bar{j}j})/2) \tag{3}$$

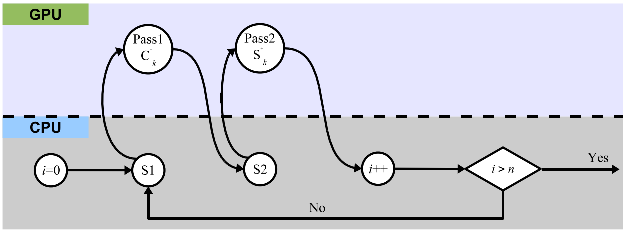


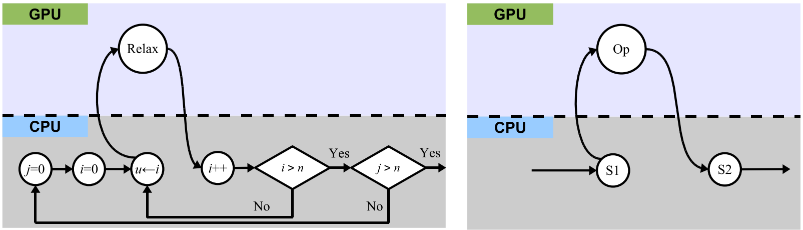
**Fig. 5.** The Closure control flow: at each cycle the CPU set parameters for the fragment program PS_Closure_C in S1 (k, and the working texture $T_0$), then the GPU executes PS_Closure_C saving the result in a texture, $T_1$. The CPU gets back the control and at stage S2 sets the parameter for fragment program PS_Closure_S (the working texture $T_1$), which is then executed by the GPU. Again the CPU gets back the control and increases the variable $i$. If $i > n$ (where $n$ is the number of variables in the program) the closure is reached, otherwise the CPU reiterates S1.

This operation can be implemented on a GPU in the following way. First the texture, $T_0$, representing the octagon that we want to close is applied $C$ using FP_CLOSURE_C in Figure 6, and it is saved in the video memory in another texture, $T_1$. Secondly at $T_1$ is applied $S$ using FP_CLOSURE_S in Figure 6 and the result is saved in $T_0$, see Figure 5 for a visualization of the control flow. This process is iterated $n$ times, where $n$ is the number of variables in the program.

```
float4 FP_Closure_C(vertexOutput IN, Sampler2D texWork_T1): COLOR{
    float2 c0= IN.t0.xy;
    if(c0.x==c0.y)
        return 0.0f;
    else{
        float4 val0,val1,tmpVal,tmpVal2,tmpVal3,tmpVal4;
        tmpVal=tex2D(texWork1,float2(k,c0.y));
        tmpVal2=tex2D(texWork1,float2(c0.x,kXOR));
        tmpVal3=tex2D(texWork1,float2(kXOR,c0.y));
        tmpVal4=tex2D(texWork1,float2(c0.x,k));
        val0=    tex2D(texWork1,c0);
        val1=    tmpVal4+tmpVal;
        val0=(val0>val1)?val1:val0;
        val1=    tmpVal2+tmpVal3;
        val0=(val0>val1)?val1:val0;
        val1=    tmpVal4+tex2D(texWork1,float2(k,kXOR))+tmpVal3;
        val0=(val0>val1)?val1:val0;
        val1=    tmpVal2+tex2D(texWork1,float2(kXOR,k))+tmpVal;
        return (val0.x>val1.x)?val1.x:val0.x;
    }};
```

```
float4 FP_Closure_S(vertexOutput IN, Sampler2D texWork_T1):
COLOR{
    float2 c0=IN.t0.xy;
    float4 val0,val1;
    val0=(tex2D(texWork_T1,float2(c0.x,XORAddress(c0.x)))+
     tex2D(texWork_T1,float2(XORAddress(c0.y),c0.y)))/2.0f;
    val1=tex2D(texWork_T1,c0);
    float4 ret=(val1<val0)?val1:val0;
    return ret>1e30?3.4e38f:ret;
};
```

**Fig. 6.** FP_Closure_C is the FP that implements the $C$ function: values k and kXOR are constant values set by CPU, kXOR is the xorred value of k. k represents the value $k$ in Equation 2. FP_CLOSURE_S implements $S$ (Equation 3).



(a) (b)

**Fig. 7.** GPU and CPU control flow: a) the control flow for the relaxation part of the Bellman-Ford Algorithm. b) the control flow of a simple operation, the CPU leaves the control to the GPU to execute the fragment program for the simple operation.

### 4.2 Emptiness Test

The emptiness test checks if an octagon $(\mathbf{m})^\bullet = \emptyset$. This happens if only if the graph of $(\mathbf{m})^\bullet$ has a cycle with a strictly negative weight [18]. A well known algorithm for detecting negative weight cycle is the Bellman-Ford algorithm.

The mapping is realized as follow. Firstly we initialize texDist the distance array, a 1D Texture, using a constant shader which returns infinity (3.4e38). The size of this texture is $2n$, where $n$ is the number of variables. Secondly we compute the first part of the algorithm, we iterate $(2n)^2$ the FP_Relax, Figure 8, that relaxes edges, see Figure 7.a for control flow. Finally we call FP_Relax_Red, Figure 8, to find out if there is negative cycle. This function marks with value 1.0 a negative cycles, otherwise it return 0.0. The result of the test is collected applying a reduction using a Maximum kernel Figure 4.

### 4.3 Simple Operators

There are some operators used in the octagon domain that require only one pass to obtain the result. This means that the fragment program for that operator is

```
float4 FP_Relax(vertexOutput IN, float u,
                Sampler1D texDist, Sampler2D texWork1): COLOR{
    float4 distU=tex1D(texDist,u);
    float4 distV=tex1D(texDist,IN.t0.y);
    float4 weight=tex2D(texWork1,float2(u,IN.t0.y));
    if(weight>=3.4e38)
        return distV;
    else{
      float4 sum=distU+weight;
      return distV>sum?sum:distV;
    }};
```

```
float4 FP_Relax_Red(vertexOutput IN,
                    Sampler1D texDist, Sampler2D texWork1): COLOR{
    float4 distU=tex1D(texDist,IN.t0.x);
    float4 distV=tex1D(texDist,IN.t0.y);
    float4 weight=tex2D(texWork1,IN.t0.xy);
    if(weight>=3.4e38)
       return 0.0;
    else
       return (distU>=(distV+weight))?1.0:0.0;
};
```

**Fig. 8.** FP_Relax is a FP that implements the relaxation for a row in the adjacency matrix. FP_Relax_Red is a FP that checks if there is a negative cycle (returns 1.0), the complete check is realized applying a max reduction operation.

called only once, in Figure 7.b the control flow between CPU and GPU is shown for simple operators. These operators are: union, intersection, test guard, and assignment.

**Union and Intersection.** Union and Intersection between octagons are both used to implement complex guards and to merge the control flow in *if else* and *loop* commands. These operators are implemented using the upper $\sqcup^{\mathbf{cDBM}}$ and lower $\sqcap^{\mathbf{cDBM}}$ bound operators [18]:

$$[\mathbf{m} \sqcap^{\mathbf{cDBM}} \mathbf{n}]_{ij} \stackrel{\text{def}}{=} \min(\mathbf{m}_{ij}, \mathbf{n}_{ij}) \tag{4}$$

$$[(\mathbf{m})^{\bullet} \sqcup^{\mathbf{cDBM}} (\mathbf{n})^{\bullet}]_{ij} \stackrel{\text{def}}{=} \max((\mathbf{m})^{\bullet}_{ij}, (\mathbf{n})^{\bullet}_{ij}) \tag{5}$$

The implementation on GPU is quite effortless, it is only required to write in the fragment program Equation 4 and Equation 5, as it is shown in Figure 7. Note that when we calculate the union operator we need to apply the closure to $\mathbf{m}$ and $\mathbf{n}$.

**Test Guard Operator.** The test guard operator model how to analyze guards in programs. The main guard tests, that can be modeled in the octagon domain, are: $v_k + v_l \leq c$, $v_k - v_l \leq c$, $-v_k - v_l \leq c$, $v_k + v_l = c$, $v_k \leq c$, and $v_k \geq c$. All these various tests can be similarly modeled by using the first test, as proved in [18]. So we will illustrate the implementation for $v_k + v_l \leq, c$ the others are similar. The octagon operator for this is defined as:

$$[\mathbf{m}_{(v_k+v_l\leq c)}]_{ij} \stackrel{\text{def}}{=} \begin{cases} \min(\mathbf{m}_{ij}, c) & \text{if } (j,i) \in \{(2k, 2l+1); (2l, 2k+1)\} \\ \mathbf{m}_{ij} & \text{elsewhere} \end{cases} \tag{6}$$

In this case, as for Union and Intersection operators, we need only to write a simple fragment program that implements Equation 6. However in order to save very costly *if-else* commands, for checking if $(j,i) \in \{(2k, 2l+1); (2l, 2k+1)\}$, we can solve this calculating the dot product between the difference vector of $\{(2k, 2l+1), (2l, 2k+1)\}$ and $(j,i)$. This operation could be heavy, but dot product is an hardware built-in function and it performs faster than executing *if-else* commands on a GPU [22], see Figure 7 for the fragment program on GPU for this operator.

**Assignment Operators.** The assignment operators model how to analyze assignments in programs. The main assignments, that can be modeled in the octagon domain, are: $v_k \leftarrow v_k + c$, $v_k \leftarrow v_l + c$ and $v_k \leftarrow e$ where $e$ is a generic expression. As for the test guards operators we will show the implementation on the GPU for the first assignment, the others are similar. Firstly we define the assignment operator for $v_k \leftarrow v_k + c$:

$$[\mathbf{m}_{(v_k \leftarrow v_k + c)}]_{ij} \stackrel{\text{def}}{=} \mathbf{m}_{ij} + (\alpha_{ij} + \beta_{ij})c \tag{7}$$

with

$$\alpha_{ij} \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } j = 2k \\ -1 & \text{if } j = (2k+1) \\ 0 & \text{elsewhere} \end{cases} \qquad \beta_{ij} \stackrel{\text{def}}{=} \begin{cases} -1 & \text{if } i = 2k \\ 1 & \text{if } i = (2k+1) \\ 0 & \text{elsewhere} \end{cases} \tag{8}$$

Again as in the case of the Union or Intersection operators we need only to write Equation 7 in the fragment program, as it is shown in Figure 9.

```
float4 FP_ASSIGN1(vertexOutput IN, Sampler2D texWork1, float c,
                      float k2, float k2add1): COLOR{
    float4 val=tex2D(texWork_T1,IN.t0);
    float alpha,beta;
    if(IN.t0.y==k2)
        alpha= 1.0f;
    else
        alpha= (IN.t0.y==k2add1)?-1.0f:0.0f;
    if(IN.t0.x==k2)
        beta=-1.0f;
    else
        beta= (IN.t0.x==k2add1)?1.0f:0.0f;
    return val+(beta+alfa)*c;};
```

```
float4 FP_TEST_GUARD1(vertexOutput IN, Sampler2D texWork1,
      float c, float2 coord1, float coord2): COLOR{
    float4 val=tex2D(texWork_T1,IN.t0);
    float2 diff;
    float ret;
    diff=IN.t0.yx-coord1;
    if(dot(diff,diff)==0.0f)
        return min(val,c):val;
    diff=IN.t0.yx-coord2;
    return (dot(diff,diff)==0.0f)?min(val,c):val;};
```

**Fig. 9.** The fragment program for the assignment $v_k \leftarrow v_k + c$ and test guard $v_k + v_l \leq c$

### 4.4 Widening

Widening is an operator that is used to speed-up the convergence in abstract interpretation [4] returning an upper approximation of the least fixpoint $\bigvee_{i\in\mathbb{N}} F^i(\mathbf{m})$ greater than $\mathbf{m}$ of an operator (predicate transformer) $F$:

$$[\mathbf{m}\nabla\mathbf{n}]_{ij} \stackrel{\text{def}}{=} \begin{cases} \mathbf{m}_{ij} & \text{if } \mathbf{n}_{ij} \leq \mathbf{m}_{ij} \\ +\infty & \text{elsewhere} \end{cases} \tag{9}$$

As it can be seen from Equation 9, the widening operator can be easily realized as a simple operator, indeed the fragment is very simple, see Figure 11. When we analyze a loop such as:

$$[l_i \ \textbf{while}\ g\ \textbf{do}\ l_j ... l_k\ \textbf{done}\ l_{k+1}]$$

where $l_i$ is a pointer to a program location we need to solve $\mathbf{m}_j = (\mathbf{m}_i \sqcup^{\mathbf{cDBM}} \mathbf{m}_k)_g$, this is done iteratively. Starting from $\mathbf{m}_i$, the octagon for location $l_i$, $\mathbf{m}_k$ can be deduced from any $\mathbf{m}_j$ using propagation. We compute the sequence $\mathbf{m}_j$:

$$\begin{cases} \mathbf{m}_{j,0} = (\mathbf{m}_i)_{(g)} \\ \mathbf{m}_{j,n+1} = \mathbf{m}_{j,n} \nabla ((\mathbf{m}_i)^{\bullet}_{(g)}) \end{cases} \tag{10}$$

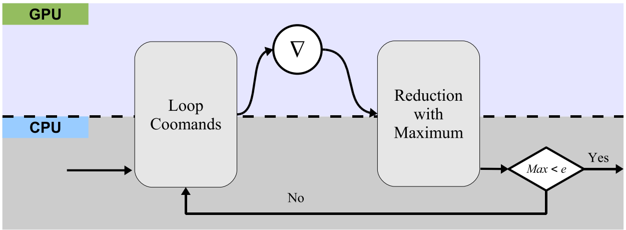


**Fig. 10.** The widening control flow: when we need to analyze a loop we proceed iteratively to the calculation of the fix point for that loop, this process needs to analyze commands in the loop, *Loop Commands* box in the flow, then we calculate the widening between $\mathbf{m}_{j,n}$ and $((\mathbf{m}_i)^{\bullet}_{(g)})$. After that we check if $\mathbf{m}_{j,n}$ and $\mathbf{m}_{j,n+1}$ represent the same octagon. This achieved calculating the maximum of the difference of them using the reduction paradigm performed in the *Reduction with Maximum* box. If the maximum is lower than $e$ a small positive value they are the same octagon.

and finally $\mathbf{m}_{k+1}$ is set equal to $((\mathbf{m}_i)^{\bullet}_{\neg g}) \sqcup^{\mathbf{cDBM}} ((\mathbf{m}_k)^{\bullet}_{\neg g})$. The calculation of the whole widening process on GPU for analyzing loops is performed in the following way: we enter in the loop and we analyze each command in the loop. After that we calculate the widening on GPU using Figure 11 between $\mathbf{m}_{j,n}$ and $((\mathbf{m}_i)^{\bullet}_{(g)})$. Finally we check if we have reached a fix point, this is realized by comparing $\mathbf{m}_{j,n+1}$, widening result, with $\mathbf{m}_{j,n}$. If it is the same octagon we reached the fix point, otherwise we need to reiterate the process. This comparison on GPU is achieved calculating the difference, $D_0$ between the result of the widening and the previous result. At this point we calculate the maximum value of $D_0$ using reduction paradigm, if the maximum is lower then a certain threshold $e$ (a small value greater than zero) the two octagons are the same otherwise they are different, all these operations can be seen summarized in Figure 10.

```
float4 FP_Widening(vertexOutput IN, Sampler2D texWork1): COLOR{
        float4 r1=tex2D(texWork1, IN.t0);
        float4 r2=tex2D(texWork2, IN.t0);
        return r2<r1?r1:3.4e38;
};
```

```
float4 FP_Top(vertexOutput IN): COLOR{
        return (IN.t0.x==IN.t0.y)?3.4e38f:0.0f;};

float4 FP_Bottom(vertexOutput IN): COLOR{
        return 0.0f;};
```

**Fig. 11.** The widening operator and the basic octagons $\top^{\mathbf{cDBM}}$ and $\bot^{\mathbf{cDBM}}$

### 4.5 Packing Data in RGBA Colors

Current GPUs can handle 4096 × 4096 2D texture size, so the maximum number of variables is 4096. However we can allow 8192 variables using pixel packing. A pixel is generally composed by four components: red, blue, green and alpha. So we can easily use these channels to allow bigger matrices, this means we treat red, green, blue, and alfa as four neighbor values in the octagon, see Figure 4.5. One advantage of RGBA packing is that we do not have to modify our fragment programs, since GPU performs vector float point arithmetic operators, assignments, and the ternary operator (test?cond1:cond2). Also we do not use

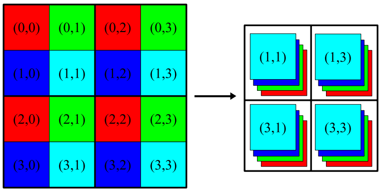


**Fig. 12.** The RGBA Packing: four neighbor values are packed in a single pixel using red, green, blue and alfa channels

the values of a octagon in the dynamic branching (*if-else*), therefore we do not have to extend the branching for each components. Another common technique is to flatten 3D textures, this means to access to a $(i, j, k)$ memory location using a $(i, j)$ address. However representing octagon bigger than $8192 \times 8192$ using 3D texture implies to use more than 64MB of video memory for each, so we can keep on the GPU system only few octagons.

### 4.6 Static Analyzer

We designed a (naive) static analyzer as presented in [6,18], in which the control flow is performed by the CPU but simple operators (union, intersection, assignments, guard checks), closure, widening, and fix point check are performed on GPU. When the interpreter starts to analyze a program, first it sets $\mathbf{m} = \top^{\mathbf{cDBM}}$ the octagon associated with the first program point, by using the simple fragment program in Figure 11. Then it interprets programs naively by applying octagon operators for the various commands and combinations of them:

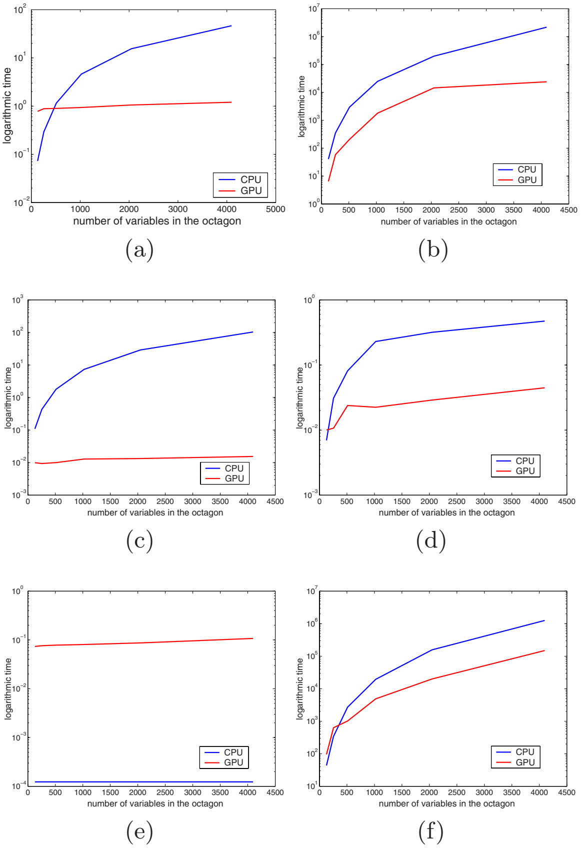
- [ $l_i$ $v_i \leftarrow e$ $l_j$ ]: we used the assignment operators;
- [ $l_i$ **if** $g$ **then** $l_j$... **else** $l_k$... **end if** $l_p$ ]: for the branch $l_j$ we apply the guard operator $g$ to the octagon $\mathbf{m}_i$) (representing point $l_i$), while for the branch $l_j$ we apply the guard operator $\neg g$ to the octagon $\mathbf{m}_i$). When the flow control merges at point $l_p$ we use the union operator:

$$\mathbf{m}_p = ((\mathbf{m}_j)^{\bullet}) \sqcup^{\mathbf{cDBM}} ((\mathbf{m}_k)^{\bullet}) \tag{11}$$

- [ $l_i$ **while** $g$ **do** $l_j$...$l_k$ **done** $l_{k+1}$ ]: we used the widening operator as presented in Section 4.4 for approximating the fixpoint.

## 5 Results

We implemented our abstract interpreter in C++, by using OpenGL [20], a successful API for computer graphics that allows GPU programming, and Cg language for GPU programming. For results we used a machine equipped with an Intel Pentium 4 D 3.2 Ghz processor, 2GB of main memory, and a GeForce 7950-GTX with 512MB of video memory. We compared the results of our interpreter with a single threaded CPU interpreter. In our experiments, with randomly generated octagons, we timed single operators such as reduction, closure, intersection, union, assignment, test guard, widening, and emptiness test. The



(a)

(b)

(c)

(d)

(e)

(f)

**Fig. 13.** Timing comparison between CPU and GPU for the octagon domain operators. We displayed timing in logarithmic scale of the time expressed in milliseconds: a) reduction operator (for checking fix point). b) closure operator. c) union (similar for intersection and widening). d) assignment. e) test guard. f) emptiness test.

results for these operators by using a single GPU, compared with a CPU, are presented in Figure 13. For each operators we reached the following speed-ups, that can be derived from Figure 13:

1. **Reduction Operator:** while the computational complexity for the CPU implementation is $\mathcal{O}(n^2)$, the one for GPU is $\mathcal{O}(\frac{n^2}{p}\log n^2)$, where $n$ is the number of variables in an octagon and $p$ is the number of FPUs. However its computational constant is lower than the one for CPU, so the speed-up is reasonable, achieving 9.98 times in average.
2. **Closure Operator:** the computational complexity is the same for both implementations, GPU and CPU, $\mathcal{O}(n^3)$. We achieved a 24.13 times speed-up in average.

3. **Emptiness Test:** the computational complexity is the same for both implementations, GPU and CPU, $\mathcal{O}(n^3)$. We achieved a 4.0 times speed-up in average. Note that the speed-up is lower than the one for the Closure Operator because we need to perform a test (FP_Relax_Red) and a reduction operation.
4. **Union Operator:** the computational complexity is the same for both implementations, GPU and CPU, $\mathcal{O}(n^2)$. We achieved a 160.5 times speed-up in average.
5. **Assignment Operator:** while the computational complexity of this operator for the CPU implementation is $\mathcal{O}(n)$ (we need to modify only two columns), the one for GPU is $\mathcal{O}(n^2)$. This is caused by SIMD nature of GPUs that needs to work on all values of a texture and not only a portion of it. For this operator we reached a lower speed-up than other operators, 6.47 times in average.
6. **Test Guard Operator:** this represents the worst operator in our implementation test. Since for the CPU implementation we need to modify only a constant number of values in the octagon, its computational complexity is $\mathcal{O}(1)$. However on GPUs we cannot modify only few values in a texture, but all values, so the complexity is $\mathcal{O}(n)$. In this case the CPU performs better than GPU with a 175.43 times speed-up in average. However it is faster to perform this operator on GPU than a computation on CPU followed by a transfer to GPU. This is because we need to transfer a big amount of data through the bus, which is typically a bottleneck in the architecture (see Figure 2).
7. **Widening Operator:** as in the case of Union Operator, the computational complexity for both implementations is $\mathcal{O}(n^2)$. We achieved a 114.7705 times speed-up in average.

| Number of variables | Reduction | Closure | Union | Assignment | Test guard | Widening | Emptiness Test |
|---|---|---|---|---|---|---|---|
| **CPU** | | | | | | | |
| 128 | 0.072928 | 40.42 | 0.095516 | 0.0069554 | 0.0001233 | 0.10912 | 43.73 |
| 256 | 0.29318 | 348.79 | 0.36542 | 0.030791 | 0.0002158 | 0.43517 | 342.10 |
| 512 | 1.1579 | 2949.90 | 1.5013 | 0.081164 | 0.0002226 | 1.7894 | 2716.42 |
| 1024 | 4.618 | 2.4863e4 | 6.8319 | 0.22949 | 0.001927 | 7.3287 | 1.9501e4 |
| 2048 | 15.491 | 1.9902e5 | 24.402 | 0.31771 | 0.0001477 | 28.997 | 1.5683e5 |
| 4096 | 46.444 | 2.172e6 | 73.212 | 0.47234 | 0.0001477 | 103.17 | 1.2546e6 |
| **GPU** | | | | | | | |
| 128 | 0.77948 | 6.4472 | 0.08497 | 0.010033 | 0.073362 | 0.099478 | 95.947 |
| 256 | 0.88522 | 58.346 | 0.097702 | 0.010693 | 0.076067 | 0.093238 | 632.03 |
| 512 | 0.89636 | 206.1 | 0.104 | 0.02383 | 0.0782 | 0.10923 | 1019.10 |
| 1024 | 0.93726 | 1804 | 0.1093 | 0.022384 | 0.080448 | 0.1288 | 4880.9 |
| 2048 | 1.0562 | 1.4470e4 | 0.11889 | 0.0289 | 0.086864 | 0.13345 | 1.9912e4 |
| 4096 | 1.2043 | 2.3951e4 | 0.18231 | 0.044535 | 0.10735 | 0.15447 | 1.4934e5 |

From the Table above the results of our implementation on CPU and GPU, the timing is expressed in second for 20 runs for each operator. As can we see, we obtain a sensible speed-up for simple operators (intersection, union, assignment, test guard, widening, emptiness test) and a speed-up around 24 times for closure operator. The bottleneck in our system is given by the test guard operator (Figure 13.e), indeed an optimized CPU implementation of this operator takes only $\mathcal{O}(1)$.

## 6 Conclusion and Future Work

We presented a new implementation of Abstract Octagon Domain on GPU, improving efficiency in time. Another advantage of our implementation is that it is fully compatible with old GPUs (3-4 years models) and not only new models. Computational complexity of the algorithm is kept the same, with exception of the operation for checking if the fix point has been reached during the analysis of loops. While the complexity of this operation on CPU is $\mathcal{O}(n^2)$ where $n$ is the number of variables, it is $\mathcal{O}(\frac{n^2}{p} \log n^2)$, where $p$ is the number of FPUs, due to the overhead in the reduction phase. The main limits of our current implementation are: the test guard operator which is linear in the number of variables and the size of a DBM, which is $8192 \times 8192$, meaning that we can model octagons with 4096 variables using the RGBA packing technique. In future work we would like to extend the size of octagons using hierarchical techniques as presented in [15]. In these techniques, a larger texture is sliced in subtextures which are addressed using a *page texture*. A page texture presents as values pointers to access to the desired subtexture. We are also interested in upgrading our implementation with upcoming Graphics Hardware. One of the main advantages of the new generation is the ability to randomly write the results of a fragment program in the video memory. Therefore there will be no more need to perform reduction to check when to stop in the widening operator or for the emptiness test, improving the complexity and performance for these operators, which represents our main bottleneck. Another advantage would be the suppression of XORAddress function, since new GPUs present integer arithmetic, saving memory and GPU performance. The new generation presents a better floating point implementation (very close to IEEE754 standard, with still some issues for handling specials) that could improve our implementation. This improved precision does not solve the unsound problems, that can be solved using interval linear forms [19]. In future work we would like to map efficiently interval linear forms on GPU.

## References

1. Blythe, D.: The direct3d 10 system. In: SIGGRAPH '06: ACM SIGGRAPH 2006 Papers, pp. 724–734. ACM Press, New York, USA (2006)
2. Bolz, J., Farmer, I., Grinspun, E., Schröoder, P.: Sparse matrix solvers on the gpu: conjugate gradients and multigrid. ACM Trans. Graph. 22(3), 917–924 (2003)
3. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for gpus: stream computing on graphics hardware. ACM Trans. Graph. 23(3), 777–786 (2004)
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California, pp. 238–252. ACM Press, New York (1977)
5. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)

6. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 84–96. ACM Press, New York, NY, USA , pp. 84–96(1978)
7. Da Graçca, G., Defour, D.: Implementation of float-float operators on graphics hardware. ArXiv Computer Science e-prints (March 2006)
8. Fan, Z., Qiu, F., Kaufman, A., Yoakum-Stover, S.: Gpu cluster for high performance computing. In: SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, Washington, DC, USA, p. 47. IEEE Computer Society Press, Los Alamitos (2004)
9. Galoppo, N., Govindaraju, N., Henson, M., Manocha, D.: Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In: Gschwind, T., Aßmann, U., Nierstrasz, O. (eds.) SC 2005. LNCS, vol. 3628, p. 3. Springer, Heidelberg (2005)
10. Govindaraju, N., Gray, J., Kumar, R., Manocha, D.: Gputerasort: high performance graphics co-processor sorting for large database management. In: SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, ACM Press, New York, pp. 325–336 (2006)
11. Horn, D.R., Houston, M., Hanrahan, P.: Clawhmmer: A streaming hmmer-search implementation. In: Gschwind, T., Aßmann, U., Nierstrasz, O. (eds.) SC 2005. LNCS, vol. 3628, p. 11. Springer, Heidelberg (2005)
12. IEEE. IEEE 754: Standard for binary floating-point arithmetic.
13. Kessenich, J., Baldwin, D., Rost, R.: The opengl shading language v.1.20 revision 8 (September 2006)
14. Kruger, J., Westermann, R.: Linear algebra operators for gpu implementation of numerical algorithms. In: SIGGRAPH '03: ACM SIGGRAPH 2003 Papers, San Diego, California, ACM Press, New York, NY, USA, pp. 908–916 (2003)
15. Lefohn, A., Sengupta, S., Kniss, J., Strzodka, R., Owens, J.: Glift: Generic, efficient, random-access gpu data structures. ACM Trans. Graph. 25(1), 60–99 (2006)
16. Mark, W., Glanville, R., Akeley, K., Kilgard, M.: Cg: a system for programming graphics hardware in a c-like language. In: SIGGRAPH '03: ACM SIGGRAPH 2003 Papers, pp. 896–907. ACM Press, New York, NY, USA (2003)
17. McCool, M., Toit, S.D., Popa, T., Chan, B., Moule, K.: Shader algebra. In: SIGGRAPH '04: ACM SIGGRAPH 2004 Papers. ACM Press, New York, NY, USA, pp. 787–795.(2004)
18. Miné, A.: The octagon abstract domain. In: AST 2001 in WCRE 2001, October 2001. IEEE, pp. 310–319. IEEE Computer Society Press, Los Alamitos (2001)
19. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 3–17. Springer, Heidelberg (2004)
20. Neider, J., Davis, T.: OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA (1993)
21. Owens, J., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A., Purcell, T.: A survey of general-purpose computation on graphics hardware. In: Eurographics 2005, State of the Art Reports, pp. 21–51 (August 2005)
22. Pharr, M., Fernando, R.: GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley, Redwood City, CA, USA (2005)
23. Venkatasubramanian, S.: The graphics card as a stream computer. In: SIGMOD-DIMACS Workshop on Management and Processing of Data Streams (2003)

# Fixpoint-Guided Abstraction Refinements[*]

Patrick Cousot[1], Pierre Ganty[2,**], and Jean-François Raskin[2]

[1] Département d'informatique, École normale supérieure
45 rue d'Ulm, 75230 Paris cedex 05, France
Patrick.Cousot@ens.fr
[2] Département d'informatique, Université Libre de Bruxelles
Campus de la Plaine, CP212, 1050 Bruxelles, Belgium
{pganty,jraskin}@ulb.ac.be

**Abstract.** In this paper, we present an abstract fixpoint checking algorithm with automatic refinement by backward completion in Moore closed abstract domains. We study the properties of our algorithm and prove it to be more precise than the counterexample guided abstract refinement algorithm (CEGAR). Contrary to several works in the literature, our algorithm does not require the abstract domains to be partitions of the state space. We also show that our automatic refinement technique is compatible with so-called acceleration techniques. Furthermore, the use of Boolean closed domains does not improve the precision of our algorithm. The algorithm is illustrated by proving properties of programs with nested loops.

## 1 Introduction

Techniques for the automatic verification of program's invariants is an active research subject since the early days of computer science. Invariant verification for a program $P$ can be reduced to a *fixpoint checking problem*: given a monotone function $post$ over sets of program states, a set of initial states $I$, and a set $S$ of states, $S$ is an *invariant* of $P$ if and only if the reachable states $\bigcup_{i \geqslant 0} post^i(I)$ from $I$ that is the least fixpoint of $\lambda X.\, I \cup post(X)$ is a subset of $S$. We call this fixpoint the forward semantics of $P$.

For fundamental reasons (undecidability of the invariant checking problem for Turing complete models of computation), or for practical reasons (limitations of the computing power of computers), the forward semantics is usually not evaluated in the domain of the function $\lambda X.\, I \cup post(X)$, the so-called *concrete domain*, but in a simpler domain of values, a so-called *abstract domain*. Abstract interpretation has been proposed in [1] as a general theory to abstract fixpoint checking problems. The design of effective abstract interpretation algorithms relies on the definition of useful abstract domains and semantics. The design of good abstractions for a programming language is a difficult and time consuming tasks. Recently, research [2,3,4] efforts have been devoted to find automatic techniques that are able to discover and refine abstract domains for a given program. This work proposes new results in this line.

[*] This research was supported the Belgian FNRS grant 2.4530.02 of the FRFC project "Centre Fédéré en Vérification" and by the project "MoVES", an Interuniversity Attraction Poles Programme of the Belgian Federal Government.

[**] Pierre Ganty is supported by the FNRS under a FRIA grant.

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 333–348, 2007.

In this paper, we propose a new *abstract algorithm* for fixpoint checking with built-in *abstract domain refinements*. The automatic refinement of abstract domains is used to improve the precision of the algorithm when it is inconclusive. Our algorithm has several properties that distinguishes it from the existing algorithms proposed in the literature. First, it computes not only overapproximations of least fixpoints but also *overapproximations of greatest fixpoints*. The two analyses improve each other: the current fixpoint is bound to use values which are more precise than the previous fixpoint. Second, it is not bound to consider refinements related to spurious abstract counterexamples. The refinement principle that we propose is guided by the abstract fixpoint computations. Our refinement method is more robust and systematic. Third, our refinement principle is compatible with acceleration techniques: acceleration techniques can be used to discover new interesting abstract values which can be used by subsequent abstract computations. This is an important characteristic as this allows us to compute new abstract values that are useful to capture the behavior of loops. This hinders the application of the CEGAR approach. Fourth, in the abstract interpretation framework the subset of concrete values given by the abstract domain is a Moore family. Intuitively it means that the set is closed for the meet operation of the concrete lattice. This property is weaker than the property enforced by the use of partitions of the state space as in so-called *predicate abstractions*. In the paper we show that requiring the use of partitions instead of Moore families does not add power to our algorithm. If it terminates using partitions then it terminates using Moore families. Fifth we show that whenever an invariant can be proved using the CEGAR approach then our algorithm is able to prove the invariant as well. And last we show that the abstract algorithm is guaranteed to terminate under various conditions like for instance the descending chain condition on the concrete domain or if the refinement adds a value for which the concrete greatest fixpoint is computable.

**Related works.** In the following pages we relate our approach with the CEGAR approach (see [5]) where the refinement is done by a backward traversal of the abstract counterexample. Recently new refinement techniques based on the proof of unsatisfiability of the counterexample emerged (see [6] and the references given there). Seen differently, the refinement picks non deterministically the new values to add to the abstract domain among a set of values defined declaratively. In our case the value is unique and defined operationally. For this reason we think that an empirical comparison would make more sense.

The abstract fixpoint checking algorithm we propose is an extension of the classical combination of forward and backward static analysis in abstract interpretation ([7] as generalized by [8]) to include abstract domain completion that is the extension of the abstract domain to avoid loss of precision in abstract fixpoints. This abstract domain completion is a backward completion in the classical sense of abstract interpretation [9] but, for efficiency, restricted to states reachable within the invariant to be checked. In [10] the authors define a restricted abstract domain completion. However since we reuse all the information computed so far our completion is much more finer than theirs. In [11] the authors consider a set of proof rules to establish invariant properties of the system and they propose abstractions to show the premises of some rule hold. Moreover they give a method to exclude spurious counterexamples based on acceleration techniques.

**Structure of the paper.** The paper is organized as follows. Sect. 2 introduces some preliminary results that are useful for the rest of the paper. In Sect. 3, we present our algorithm and prove its main properties related to correctness and termination, we also show that our approach can be easily combined with acceleration techniques. Sect. 4 compares our algorithm to the CEGAR approach and predicate abstraction. Sect. 5 concludes the paper and proposes some future works.

## 2 Preliminaries

**Notations and notions of lattice theory.** We use Church's lambda notation (so that $F$ is $\lambda X.\, F(X)$) and use the composition operator $\circ$ on functions given by $(f \circ g) = \lambda X.\, f(g(X))$. Let $X$ be any set and let $f \in X \mapsto X$ be a function on this set. The reflexive transitive closure $f^*$ of a function $f$ such that its domain and co-domain coincide is given by $\bigcup_{i \geq 0} f^i$ where $f^0$ is the identity and $f^{i+1} = f^i \circ f$. The reflexive transitive closure $R^*$ of a relation $R$ is defined in the same way. A function $f$ on a complete lattice is said to be *additive* (resp. *coadditive*) if $f$ distributes the join (resp. the meet) operator. Given two functions $f, g$ on a poset $(L, \subseteq)$, we define the pointwise comparison $\dot{\subseteq}$ between functions as follows: $\lambda x.\, f(x) \mathrel{\dot{\subseteq}} \lambda x.\, g(x)$ iff $\forall y \in L\colon f(y) \subseteq g(y)$. Given a set $S$, $\wp(S)$ denote the set of all subsets of $S$. Sometimes we write $s$ instead of the singleton $\{s\}$.

We denote by $\mathit{lfp}(f)$ and $\mathit{gfp}(f)$, respectively, the least and greatest fixpoint, when they exist, of a function $f$ on a poset. The well-known Knaster-Tarski's theorem states that any monotone function $f \in L \mapsto L$ on a complete lattice $\langle L, \leqslant, \wedge, \vee, \top, \bot \rangle$ admits a least fixpoint and the following characterization holds: $\mathit{lfp}(f) = \bigwedge \{x \in L \mid f(x) \leqslant x\}$. Dually, $f$ also admits a greatest fixpoint and the following characterization holds: $\mathit{gfp}(f) = \bigvee \{x \in L \mid x \leqslant f(x)\}$.

**Transition systems and predicate transformers.** A *transition system* is a 3-tuple $\mathcal{T} = (C, I, T)$ where $C$ is the set of *states*, $I \subseteq C$ is the subset of *initial states*, and $T \subseteq C \times C$ is the *transition relation*. Often, we write $s \rightarrow s'$ if $(s, s') \in T$, $s \rightarrow^* s'$ if $(s, s') \in T^*$ and $s \rightarrow^k s'$ if $(s, s') \in T^k$ for $k \in \mathbb{N}$.

To manipulate sets of states, we use *predicate transformers*. The *forward image operator* is a function that given a relation $T' \subseteq C \times C$ and a set of states $C' \subseteq C$, returns the set $\mathit{post}[T'](C') = \{c' \in C \mid \exists c \in C' : (c, c') \in T'\}$. When the forward image is used with the transition relation $T$, it is called the *post operator* and it returns, given a set of states $C'$ all its one step successors in the transition system, we simply write it $\mathit{post}(C')$. The *backward image operator* is a function given a relation $T' \subseteq C \times C$ and set of states $C' \subseteq C$, returns the set $\widetilde{\mathit{pre}}[T'](C') = \neg \mathit{pre}[T'](\neg C') = \neg \mathit{post}[{T'}^{-1}](\neg C')$ $= \{c \in C \mid \forall c'\colon (c, c') \in T' \Rightarrow c' \in C'\}$. When the backward image operator is used with the transition relation $T$, it is called the *unavoidable operator* and it returns, given a set of states $C'$ all the states which have all their successors in the set $C'$, we simply write it $\widetilde{\mathit{pre}}(C')$.

Given a set $I$ of states the set of *reachable states* is given by the following least fixpoint $\mathit{lfp}^{\subseteq} \lambda X.\, I \cup \mathit{post}[T](X)$. As shown in [12], this fixpoint coincides with $\mathit{post}[T^*](I)$ also written $\mathit{post}^*(I)$ when the transition relation is clear from the context. So a state $s$ is said to be reachable if $s \in \mathit{post}^*(I)$. Dually, given a set $S$ of states, the set of states that *are stuck in* $S$ (or also that *cannot escape from* $S$) is given by the following greatest fixpoint $gfp^{\subseteq}\lambda X.\, S \cap \widetilde{pre}[T](X)$. As shown in [12], this fixpoint coincides with $\widetilde{pre}[T^*](S)$ also written $\widetilde{pre}^*(S)$ when the transition relation is clear from the context.

Given two sets $I, Z$ of states we call $lfp^{\subseteq}\lambda X.\, (I \cup post(X)) \cap Z$ the set of *reachable states within* $Z$. Finally given a set $S$ of states, the set of *states that cannot escape from* $S$ *in less than* 1 *steps* is given by $S \cap \widetilde{pre}(S)$.

**Abstract interpretation.** We use abstract interpretation to abstract the semantics of transition systems. We assume standard abstract interpretation where, *concrete* and *abstract domains*, $L$ given by $\wp(C)$ and $A$, are Boolean complete lattice $\langle L, \subseteq, \cap, \cup, C, \emptyset, \neg\rangle$ and complete lattice $\langle A, \sqsubseteq, \sqcap, \sqcup, \top, \bot\rangle$, respectively. The two lattices are related by abstraction and concretization maps $\alpha$ and $\gamma$ forming a *Galois connection* $\forall c \in L : \forall a \in A : \alpha(c) \sqsubseteq a \Leftrightarrow c \subseteq \gamma(a)$ [7]. We write this fact as follows: $\langle L, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq\rangle$, or simply $\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}$ when the concrete and abstract domains are clear from the context. We recall here a well-known (see [13] for instance) Galois connection that will be used later on: given a transition system $(C, I, T)$ we have $\langle C, \subseteq\rangle \underset{\lambda X.post[T](X)}{\overset{\lambda X.\widetilde{pre}[T](X)}{\leftrightarrows}} \langle C, \subseteq\rangle$. In this paper, we use a family of finite abstract domains that are subset of $A$.

**Definition 1 (Family of abstract domains).** *Let* $\{A_i\}_{i\in I}$ *be a family of finite sets such that:* $(i)$ $A = \bigcup_{i\in I} A_i$, $(ii)$ $\langle A_i, \sqsubseteq\rangle$ *is a complete lattice, and* $(iii)$ $\exists \alpha_i \colon \langle L, \subseteq\rangle \underset{\alpha_i}{\overset{\gamma}{\leftrightarrows}} \langle A_i, \sqsubseteq\rangle$.

Given an abstract domain $A_i$, we write $\gamma(A_i)$ for the subset of concrete sets $X \in L$ that can be represented by abstract values in $A_i$.

The set $\gamma(A_i) \subseteq L$ of concrete values that the abstract domain represents must be closed by intersection if there is a Galois connection between $A_i$ and $L$. Our abstract domains are thus Moore closed. This notion, and the stronger notion of Boolean closure are defined as follows.

**Definition 2 (Moore and Boolean closure).** *A finite subset* $X \subseteq L$ *is said to be:*

- Moore closed *iff* $\forall x_1, x_2 \in X : x_1 \wedge x_2 \in X$ *and* $X$ *contains the topmost element of* $L$. *We define the function* $\lambda X.\, \mathcal{M}(X)$ *which returns the Moore closure of its argument, i.e. the smallest set* $M \subseteq L$ *such that* $X \subseteq M$ *and* $M$ *is Moore closed*[1].
- Boolean closed *iff* $\forall x_1, x_2 \in X$: $(i)$ $x_1 \wedge x_2 \in X$, $(ii)$ $x_1 \vee x_2 \in X$, *and* $(iii)$ $C \setminus x \in X$. *We define the function* $\lambda X.\, \mathcal{B}(X)$ *which returns the Boolean closure of its argument, i.e. the smallest set* $B$ *such that* $X \subseteq B$ *and* $B$ *is Boolean closed.*

Let $P = \{p_1, p_2, \dots, p_n\}$ be a set of predicates and let $[\![p_i]\!] \subseteq C$ be the subset of states that satisfy the predicate $p_i$. The set of predicates $P$ implicitly defines a Boolean closed abstract domain, noted $\mathcal{A}_P$, such that $\gamma(\mathcal{A}_P) \subseteq L$ is the smallest set which is Boolean closed and contains the sets $\{[\![p]\!] \mid p \in P\}$, i.e. $\gamma(\mathcal{A}_P) = \mathcal{B}(\{[\![p]\!] \mid p \in P\})$. The elements of $\mathcal{A}_P$ are equivalent to propositional formulas built from the predicates in $P$.

[1] A Moore closed set is also called a Moore family.

Elements of $\mathcal{A}_P$ can also be viewed as union of equivalence classes of states: two states $c_1, c_2 \in C$ are equivalent whenever they satisfy exactly the same subset of predicates in $P$.

The following Lemma contains well-known results of abstract interpretation that we recall here so that the paper is self contained. We refer the interested reader to [14] and the references given there for more details.

**Lemma 1.** *Let $I, S, Z \in L$ be sets of states. Given an abstract domain $A_i$, we define $\mathcal{R}$, resp. $\mathcal{S}$, to be the abstract forward, resp. backward, semantics on $A_i$ as $\mathit{lfp}^{\sqsubseteq}\lambda X.\, \alpha_i((I \cup \mathit{post}(\gamma(X))) \cap Z)$, resp. $\mathit{gfp}^{\sqsubseteq}\lambda X.\, \alpha_i(S \cap \widetilde{\mathit{pre}}(\gamma(X)))$.*

$$\left.\begin{array}{r} \mathit{lfp}^{\subseteq}\lambda X.\, (I \cup \mathit{post}(X)) \cap Z \subseteq \gamma(\mathcal{R}) \\ \mathit{gfp}^{\subseteq}\lambda X.\, S \cap \widetilde{\mathit{pre}}(X) \subseteq \gamma(\mathcal{S}) \end{array}\right\} \text{ We call this inclusion the overapproximation of the abstract semantics.}$$

**The Fixpoint Checking Problem.**
**Instance:** a transition system $(C, I, T)$ and a set of states $S \subseteq C$.
**Question:** Does the inclusion $\mathit{lfp}^{\subseteq}\lambda X.\, I \cup \mathit{post}(X) \subseteq S$ holds ?

## 3 Abstract Fixpoint Checking Algorithm

Alg. 1 has been inspired and is a generalization of what we have done previously in [15,16,17]. We review here its main characteristics.

It computes overapproximations of *least* and *greatest* fixpoints. Line 3 computes an abstract least fixpoint. As we will see in Prop. 1, when executed on a positive instance of the fixpoint checking problem, every set $\gamma(\mathcal{R}_i)$ overapproximates the reachable states of the transition system. Line 7 computes an abstract greatest fixpoint. As we will see in Lem. 2, and Lem. 3, $\gamma(\mathcal{S}_i)$ underapproximates the set of states that cannot escape from $S$ in less than $i + 1$ steps. As we can see from line 3 and line 7, the two fixpoints share all the information that has been computed so far. In fact the abstract least fixpoint of line 3 overapproximates the reachable states within $Z_i$ which gathers all the information computed so far. Similarly, the abstract greatest fixpoint of line 7 starts with the least fixpoint computed previously. Parts of the state space that have already been proved unreachable within $S$ or stuck in $S$ are not explored during the next iterations.

The refinement that we propose is applied on the entire abstract fixpoint and is not bound to individual counterexamples. The value $Z_i$ contains states that cannot escape from $\gamma(S_i)$ in one step, all concrete states that are stuck within $S$ have this property. So, this set is interesting as it adds information about concrete states in the abstract domain, this information will be used by subsequent abstract fixpoint computation. We will see later in the paper that line 9 can be modified in a way to incorporate information computed by acceleration techniques. The results that we first prove with line 9 are still valid when accelerations are used. The possibility of combining our algorithm with acceleration techniques is very interesting as accelerations may allow to discover interesting abstract values related to loops in programs. Loops usually hinder the application of the CEGAR approach.

In line 10 we see that the new value $Z_{i+1}$ computed at line 9 is added to the set of values the current abstract domain $A_i$ can represent (this set is $\gamma(A_i)$). The new

**Algorithm 1.** The abstract fixpoint checking algorithm

```
   Data: An instance of the fixpoint checking problem such that $I \subseteq S$ and an
         abstract domain $A_0$ such that $S \in \gamma(A_0)$
1  $Z_0 = S$
2  for $i = 0, 1, 2, 3, \ldots$ do
3      Compute $\mathcal{R}_i$ given by $\mathit{lfp}^{\sqsubseteq} \lambda X.\, \alpha_i\Big((I \cup \mathit{post}(\gamma(X))) \cap Z_i\Big)$
4      if $\alpha_i(I \cup \mathit{post}(\gamma(\mathcal{R}_i))) \sqsubseteq \alpha_i(Z_i)$ then
5          return OK
6      else
7          Compute $\mathcal{S}_i$ given by $\mathit{gfp}^{\sqsubseteq} \lambda X.\, \alpha_i\Big(\gamma(\mathcal{R}_i) \cap \widetilde{\mathit{pre}}(\gamma(X))\Big)$
8          if $\alpha_i(I) \sqsubseteq \mathcal{S}_i$ then
9              Let $Z_{i+1} = \gamma(\mathcal{S}_i) \cap \widetilde{\mathit{pre}}(\gamma(\mathcal{S}_i))$
10             Let $A_{i+1}$ be s.t. $\gamma(A_{i+1}) = \mathcal{M}\big(\{Z_{i+1}\} \cup \gamma(A_i)\big)$
11         else
12             return KO
13         end
14     end
15 end
```

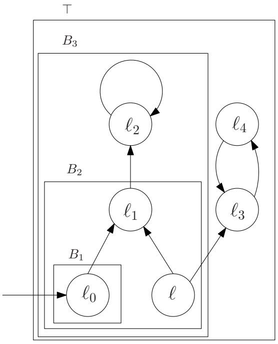
abstract domain is given by $A_{i+1}$. It is worth pointing that we actually add more than the single value $Z_{i+1}$ to the abstract domain since working in the framework of abstract interpretation requires that $\gamma(A_{i+1})$ is a Moore family. We will see later that Moore closure is sufficiently powerful in the following precise sense: considering the Boolean closure instead does not improve the precision of our algorithm. This interesting result is established in Th. 2. This contrasts with several approaches in the literature that use predicate abstraction which induce more complex Boolean closed domains. The most precise abstract *post* operation is usually more difficult to compute on Boolean closed domains.

Our algorithm also enjoys nice termination properties. Prop. 6 shows that our algorithm terminates whenever the concrete domain enjoys the descending chain condition. This result allows us to conclude that our algorithm will always terminate for the important class of Well-structured transition systems [18,19], see [16,17] for the details. Th. 1 of Sect. 4 also shows that whenever CEGAR terminates, then our algorithm terminates. We also establish in Prop. 5 that whenever our algorithm is submitted a negative instance, it always terminates.

Finally it is worth pointing out that all the operations in the algorithm, with the exception of the refinement operation of line 9, are abstract operations, and the only concrete operation is used outside of any of the fixpoint computations.

Before giving a formal characterization of Alg. 1, let us give more insights by running the algorithm on a toy example.

*Example 1.* The toy example is a finite state system given at Fig. 1. The set of states given by the initial abstract domain are given by the boxes. We submit to our algorithm



$\mathcal{R}_0 = B_3$ line 3

$\alpha_0(I \cup post(\gamma(B_3))) \not\sqsubseteq Z_0$ (so not “OK”) line 4

$\mathcal{S}_0 = B_3$ line 7

$\alpha_0(I) \sqsubseteq \mathcal{S}_0$ (cannot say “KO”) line 8

$Z_1 = \gamma(B_3) \cap \widetilde{pre}(\gamma(B_3))$ line 9

$= \{\ell_0, \ell_1, \ell_2\}$

The new domain is $A_1 = A_0 \cup \{B_4, B_5\}$ line 10

with $\gamma(B_4) = Z_1$ and $\gamma(B_5)$ = $Z_1 \cap \gamma(B_2)$

= $\{\ell_0, \ell_1\}$

$\mathcal{R}_1 = B_4$ line 3

$\alpha_1(I \cup post(\gamma(B_4))) \sqsubseteq Z_1$ line 4

Alg. 1 terminates saying “OK”

**Fig. 1.** A finite state system and the result of evaluating Alg. 1 on it

the following positive instance of the fixpoint checking problem where $A_0 = \{B_1, B_2, B_3, \top\}$, $I$ = $\{\ell_0\}$, and $S$ = $\gamma(B_3)$. So note that $Z_0 = \gamma(B_3) = S$. In the right side of Fig. 1 the algorithm is executed step by step. Since the fixpoints converge in very few steps we invite the interested reader to verify them by hand.

### 3.1 Correctness of the Algorithm

In what follows we assume that Alg. 1 reaches enough iteration to compute the sets appearing in the statements. For instance, if $\gamma(\mathcal{R}_i)$ appears in the statement then the algorithm has not yet concluded at iteration $i - 1$ or if $Z_{i+1}$ appears in the statement then the algorithm has not yet concluded at iteration $i$.

We start with a technical lemma that states that our algorithm computes sets of states that are decreasing.

**Lemma 2.** *In Alg. 1 we have*

$$Z_{i+1} \subseteq \gamma(\mathcal{S}_i) \subseteq \gamma(\mathcal{R}_i) \subseteq Z_i \subseteq \cdots \subseteq Z_1 \subseteq \gamma(\mathcal{S}_0) \subseteq \gamma(\mathcal{R}_0) \subseteq Z_0 \subseteq S.$$

The next proposition characterizes the sets of states that are computed by the algorithm in the presence of positive instances.

**Proposition 1.** *In Alg. 1, if $post^*(I) \subseteq S$ then $post^*(I) \subseteq \gamma(\mathcal{R}_i)$ for any $i \in \mathbb{N}$.*

*Proof.* Our proof is by induction on $i$.

**Base case.** Lem. 1 tells us that $\gamma(\mathcal{R}_0)$ overapproximates the following least fixpoint $\mathit{lfp}^{\subseteq}\lambda X.\,(I \cup post(X)) \cap S$. Provided the system respects the invariant $S$ (i.e. $post^*(I) \subseteq S$), this fixpoint is equal to $\mathit{lfp}^{\subseteq}\lambda X.\,(I \cup post(X))$. So, $post^*(I) \subseteq \gamma(\mathcal{R}_0)$.

**Inductive case.** For the inductive case we prove the contrapositive. Suppose that there exists $s \in post^*(I)$ and $s \notin \gamma(\mathcal{R}_i)$. We recall Lem. 2 which shows that $\gamma(\mathcal{R}_{i-1}) \supseteq \gamma(\mathcal{S}_{i-1}) \supseteq Z_i \supseteq \gamma(\mathcal{R}_i)$. We now consider several cases.

1. $s \notin \gamma(\mathcal{R}_{i-1})$. Then by induction hypothesis, $\mathit{post}^*(I) \nsubseteq S$ and we are done.
2. $s \in \gamma(\mathcal{R}_{i-1})$ and $s \notin \gamma(\mathcal{S}_{i-1})$. We conclude from Lem. 1 that $\gamma(\mathcal{S}_{i-1})$ overapproximates the states stuck in $\gamma(\mathcal{R}_{i-1})$. Since $s \notin \gamma(\mathcal{S}_{i-1})$ there exists a state $s'$ such that $s \to^* s'$ and $s' \notin \gamma(\mathcal{R}_{i-1})$. First, note that as $s \in \mathit{post}^*(I)$, we conclude that $s' \in \mathit{post}^*(I)$. But as $s' \notin \gamma(\mathcal{R}_{i-1})$, we know that $\mathit{post}^*(I) \nsubseteq \gamma(\mathcal{R}_{i-1})$ and by induction hypothesis we conclude that $\mathit{post}^*(I) \nsubseteq S$.
3. $s \in \gamma(\mathcal{R}_{i-1})$, $s \in \gamma(\mathcal{S}_{i-1})$ and $s \notin Z_i$. We conclude from the definition of $Z_i$ which is given by $\gamma(\mathcal{S}_{i-1}) \cap \widetilde{\mathit{pre}}(\gamma(\mathcal{S}_{i-1}))$ that there exists $s' \notin \gamma(\mathcal{S}_{i-1})$ such that $s \to s'$. Either $s' \notin \gamma(\mathcal{R}_{i-1})$ or $s' \in \gamma(\mathcal{R}_{i-1})$ and by the previous case, we know that $s' \to^* s''$ and $s'' \notin \gamma(\mathcal{R}_{i-1})$. In the two cases, we conclude that $\mathit{post}^*(I) \nsubseteq \gamma(\mathcal{R}_{i-1})$ and by induction hypothesis that $\mathit{post}^*(I) \nsubseteq S$.
4. $s \in \gamma(\mathcal{R}_{i-1})$, $s \in \gamma(\mathcal{S}_{i-1})$, $s \in Z_i$, and $s \notin \gamma(\mathcal{R}_i)$. By overapproximation of the abstract semantics, we know that $s$ is not reachable from $I$ within $Z_i$. Otherwise stated, all paths starting form $I$ and ending in $s$ leaves $Z_i$. As $s$ is reachable from $I$, we know that there exists some $s' \notin Z_i$ which is reachable form $I$. We can apply the same reasoning as above and conclude that $\mathit{post}^*(I) \nsubseteq S$. □

We are now in position to prove that, when the algorithm terminates and returns OK, it has been submitted a positive instance of the fixpoint checking problem, and when the algorithm terminates and returns KO, it has been submitted a negative instance of the fixpoint checking problem.

**Proposition 2 (Correctness – positive instances).** *If Alg. 1 says "OK" then we have* $\mathit{post}^*(I) \subseteq S$.

*Proof.*

$$
\begin{array}{ll}
\text{Algorithm says "OK"} & \\
\Leftrightarrow \alpha_i(I \cup \mathit{post}(\gamma(\mathcal{R}_i))) \sqsubseteq \alpha_i(Z_i) & \text{line 4} \\
\Leftrightarrow \alpha_i(I) \sqsubseteq \alpha_i(Z_i) \;\&\; \alpha_i \circ \mathit{post} \circ \gamma(\mathcal{R}_i) \sqsubseteq \alpha_i(Z_i) & \alpha_i \text{ additivity} \\
\Leftrightarrow I \sqsubseteq \gamma \circ \alpha_i(Z_i) \;\&\; \mathit{post}(\gamma(\mathcal{R}_i)) \sqsubseteq \gamma \circ \alpha_i(Z_i) & \underset{\alpha_i}{\overset{\gamma}{\leftrightarrows}} \\
\Leftrightarrow I \subseteq Z_i \;\&\; \mathit{post}(\gamma(\mathcal{R}_i)) \subseteq Z_i & Z_i \in \gamma(A_i) \text{ line 10}
\end{array}
$$

Then,

$$
\begin{array}{ll}
\alpha_i((I \cup \mathit{post}(\gamma(\mathcal{R}_i))) \cap Z_i) \sqsubseteq \mathcal{R}_i & \text{def. of } \mathcal{R}_i\text{, prop. of } \mathit{lfp} \\
\Leftrightarrow (I \cup \mathit{post}(\gamma(\mathcal{R}_i))) \cap Z_i \subseteq \gamma(\mathcal{R}_i) & \underset{\alpha_i}{\overset{\gamma}{\leftrightarrows}} \\
\Rightarrow I \cup \mathit{post}(\gamma(\mathcal{R}_i)) \subseteq \gamma(\mathcal{R}_i) & I \subseteq Z_i \;\&\; \mathit{post}(\gamma(\mathcal{R}_i)) \subseteq Z_i \\
\Rightarrow \mathit{lfp}^{\subseteq} \lambda X.\, I \cup \mathit{post}(X) \subseteq \gamma(\mathcal{R}_i) & \text{prop. of } \mathit{lfp} \\
\Rightarrow \mathit{post}^*(I) \subseteq S & \gamma(\mathcal{R}_i) \subseteq S \text{ by Lem. 2}
\end{array}
$$

□

**Proposition 3 (Correctness – negative instances).** *If Alg. 1 says "KO" then we have* $\mathit{post}^*(I) \nsubseteq S$.

*Proof.* If at iteration $i$ the algorithm says "KO" then we find that $\alpha_i(I) \not\sqsubseteq \mathcal{S}_i$ (line 8) which is equivalent to $I \not\subseteq \gamma(\mathcal{S}_i)$ by $\underset{\alpha_i}{\overset{\gamma}{\leftrightarrows}}$. We conclude from Lem. 2 that $\gamma(\mathcal{R}_{i+1}) \subseteq \gamma(\mathcal{S}_i)$, hence that $I \not\subseteq \gamma(\mathcal{R}_{i+1})$ and finally that $post^*(I) \not\subseteq S$ using the contrapositive of Prop. 1. □

*Remark 1.* The proofs of the above results remain correct if in line 9 of Alg. 1 instead of $\lambda X.\, \widetilde{pre}[T](X)$ we take $\lambda X.\, \widetilde{pre}[R](X)$ where $R \subseteq T^*$. In fact for correction to hold $Z_{i+1}$ must be such that $\widetilde{pre}[T^*](\gamma(\mathcal{S}_i)) \subseteq Z_{i+1} \subseteq \gamma(\mathcal{S}_i)$. Later we will see how we can benefit from acceleration techniques which build a relation $R$ such that $T \subseteq R \subseteq T^*$. This alternative refinement using $R$ including $T$ yields to stronger termination properties of the algorithm.

### 3.2 Termination of the Algorithm

To reason about the termination of the algorithm, we need the following technical proposition and its corollary.

**Proposition 4.** *In Alg. 1 the following holds:*

1. *if $Z_{i+1} = Z_i$ then $post(Z_i) \subseteq Z_i$;*
2. *if $I \not\subseteq Z_i$ then the algorithm terminates at iteration $i$ and returns "KO";*
3. *if $I \cup post(Z_i) \subseteq Z_i$ then the algorithm terminates at iteration $i$ and return "OK".*

**Corollary 1.** *In Alg. 1, if $Z_i = Z_{i+1}$ then the algorithm terminates.*

Alg. 1 terminates when submitted a negative instance as proved below in Lem. 3 and Prop. 5.

**Lemma 3.** *In Alg. 1, $\gamma(\mathcal{R}_i)$ underapproximates the set $\widetilde{pre}[\bigcup_{j=0}^{i} T^j](S)$ of states which cannot escape from $S$ in less than $i+1$ steps.*

*Proof.* The result is shown by induction on the number $i$ of steps. For the base case, Lem. 2 shows that $\gamma(\mathcal{R}_0) \subseteq S = \widetilde{pre}[T^0](S)$. For the inductive case,

$$\begin{aligned}
\widetilde{pre}[\bigcup_{j=0}^{i+1} T^j](S) &= \widetilde{pre}[\bigcup_{j=0}^{i} T^j \cup \bigcup_{j=1}^{i+1} T^j](S) && \text{def. } \cup \\
&= \widetilde{pre}[\bigcup_{j=0}^{i} T^j](S) \cap \widetilde{pre}[T](\widetilde{pre}[\bigcup_{j=0}^{i} T^j](S)) && \text{def. } \widetilde{pre} \\
&\supseteq \gamma(\mathcal{R}_i) \cap \widetilde{pre}[T](\gamma(\mathcal{R}_i)) && \text{ind. hyp.} \\
&\supseteq \gamma(\mathcal{S}_i) \cap \widetilde{pre}[T](\gamma(\mathcal{S}_i)) && \text{by Lem. 2} \\
&= Z_{i+1} && \text{by line 9} \\
&\supseteq \gamma(\mathcal{R}_{i+1}) && \text{by Lem. 2}
\end{aligned}$$

□

**Proposition 5.** *If $post^*(I) \not\subseteq S$ then Alg. 1 terminates.*

*Proof.* Hypothesis shows that there exists states $s, s'$ and a value $k \in \mathbb{N}$ such that $s \in I$, $s' \notin S$ and $s \to^k s'$. Lem. 3 shows that $\gamma(\mathcal{R}_{k-1}) \subseteq \bigcap_{j=0}^{k} \widetilde{pre}[T^j](S)$. So we conclude from above that $I \not\subseteq \bigcap_{j=0}^{k} \widetilde{pre}[T^j](S)$, hence that $I \not\subseteq \gamma(\mathcal{R}_{k-1})$ by transitivity and finally that $I \not\subseteq Z_k$ by Lem. 2. The last step uses Prop. 4.2 to show that the algorithm terminates. □

The following proposition states that our algorithm terminates under the descending chain condition in the concrete domain.

**Proposition 6.** *If there exists a poset $Y \subseteq L$ such that the descending chain condition holds on $\langle Y, \subseteq \rangle$ and $Z_i \in Y$ for all $i \in \mathbb{N}$ then Alg. 1 terminates.*

*Proof.* We prove the contrapositive. Assume the algorithm does not terminate. We thus obtain that $Z_0 \supset Z_1 \supset \cdots \supset Z_n \supset \cdots$ by Cor. 1 and Lem. 2 which contradicts the existence of a poset satisfying the above hypothesis. □

Below Prop. 7 establishes a stronger termination result of our algorithm which states that if the algorithm computes a value $Z_i$ from which the evaluation of the greatest fixpoint $\mathit{gfp}^{\subseteq} \lambda X.\, Z_i \cap \widetilde{pre}(X)$ terminates after a finite number of iterations then our algorithm terminates. We use classical fixpoint evaluation techniques to compute the set $\mathit{gfp}^{\subseteq} \lambda X.\, Z_i \cap \widetilde{pre}(X)$. First we start with the set $Z_i$ and then we remove the states that escape from $Z_i$ in 1 step. The set obtained is formally given by $Z_i \cap \widetilde{pre}(Z_i)$. Then we iterate this process until no state is removed.

**Proposition 7.** *If in Alg. 1 there is a value for $i$ such that $\mathit{gfp}^{\subseteq} \lambda X.\, Z_i \cap \widetilde{pre}(X)$ stabilizes after a finite number of steps, then Alg. 1 terminates.*

### 3.3 Termination of the Algorithm Enhanced by Acceleration Techniques

In this section we will study an enhancement of Alg. 1 which relies on acceleration techniques (we refer the interested reader to [20] and the references given there). Roughly speaking, acceleration techniques allow us to compute underapproximations of the transitive closure of some binary relation as, for instance the transition relation.

Assume we are given some binary relation $R$ such that $T \subseteq R \subseteq T^*$. The enhancement we propose replaces line 9 (viz. $Z_{i+1} = \gamma(\mathcal{S}_i) \cap \widetilde{pre}[T](\gamma(\mathcal{S}_i))$) by the following: $Z_{i+1} = \gamma(\mathcal{S}_i) \cap \widetilde{pre}[R](\gamma(\mathcal{S}_i))$. The definition of $R$ suggests that the value added using $R$ should be at least as precise as the one given using $T$. A very favorable situation is when $R$ equals $T^*$ but Prop. 7 is not applicable at any iteration. We conclude from $Z_1 = \mathit{gfp}^{\subseteq} \lambda X.\, \gamma(\mathcal{S}_0) \cap \widetilde{pre}(X)$ that $\mathit{post}(Z_1) \subseteq Z_1$ by $\underset{\mathit{post}}{\overset{\widetilde{pre}}{\leftrightarrows}}$, hence that the enhanced algorithm terminates at iteration where $i = 1$ by Prop. 4 while the normal algorithm might not since Prop. 7 is never applicable. Below we illustrate this situation using a toy example.

*Example 2.* Fig. 2 shows a two counters automaton and its associated semantics. The domain of the counters is the set of integers. In the automaton $x, y$ refer to the current value of the counters while $x', y'$ refer to the next value (namely the value after firing the transition). Transition $t_1$ is given by a simultaneous assignment. Discs depicts some reachable states, which are given by $\{(x, y) \mid y \leqslant x \mathbin{\&} 0 \leqslant x\}$. We will submit to Alg. 1 a positive instance of the fixpoint checking problem such that $I$ and $S$ are given by $\{(0, 0)\}$ and $\{(x, y) \mid y \neq x + 1\}$ respectively. Our initial abstract domain $A_0$ is such that $\gamma(A_0) = \mathcal{M}(S)$.

It is routine to check $\mathcal{R}_0$, computed at line 3, is such that $\gamma(\mathcal{R}_0) = S$, hence that the test of line 4 fails. It follows that we have to compute $\mathcal{S}_0$ given at line 7. Let $X^\delta, \delta$ be the sequence of iterates for $\lambda X.\, \alpha_0(\gamma(\mathcal{R}_0) \cap \widetilde{pre}[T](\gamma(X)))$ which converges to $\mathcal{S}_0$. First let us compute

$$\begin{aligned}
& S \cap \widetilde{pre}[t_2](S) & \\
&= S \cap \neg \circ pre[t_2] \circ \neg(S) & \text{def. of } \widetilde{pre} \\
&= S \cap \neg \circ pre[t_2](\{(x, y) \mid y = x + 1\}) & \text{def. of } \neg, S \\
&= S \cap \neg(\{(x, y) \mid y = x + 2\}) & \text{see Fig. 2} \\
&= S \cap \{(x, y) \mid y \neq x + 2\} & \\
&= \{(x, y) \mid y \neq x + 1\} \cap \{(x, y) \mid y \neq x + 2\} & \text{def. of } S
\end{aligned}$$

We now turn to the evaluation of the *gfp*.

$$\begin{aligned}
X^0 &= \top & \\
X^1 &= \alpha_0(\gamma(\mathcal{R}_0) \cap \widetilde{pre}[T](\gamma(X_0))) & \\
&= \alpha_0(S) & \gamma(\mathcal{R}_0) = S, \top \subseteq \widetilde{pre}[T](\top) \\
&= S & S \in \gamma(A_0) \\
X^2 &= \alpha_0(S \cap \widetilde{pre}[T](\gamma(X_1))) & \\
&= \alpha_0\big(S \cap \widetilde{pre}[t_1](\gamma(X_1)) \cap \widetilde{pre}[t_2](\gamma(X_1))\big) & \text{def. } \widetilde{pre} \\
&= \alpha_0\big(S \cap \widetilde{pre}[t_2](\gamma(X_1))\big) & S \cap \widetilde{pre}[t_1](S) = S
\end{aligned}$$

By above we find that $\alpha_0(S \cap \widetilde{pre}[t_2](S)) = S$, hence that $\gamma(\mathcal{S}_0) = S$. Since the test of line 8 succeeds the next step (line 9) is to compute $Z_1$. We use acceleration techniques to compute $Z_1$ for otherwise the algorithm does not converge. Without resorting to acceleration techniques each $Z_i$ escapes from $S$ in $i+1$ steps by firing transition $t_2$. This clearly indicates that the CEGAR approach considers counterexamples of increasing length and thus fails on this toy example. By considering the limit instead of the $Z_i$'s we obtain a value that is stuck in $S$. That value stuck in $S$ can be obtained using acceleration techniques as shown below.

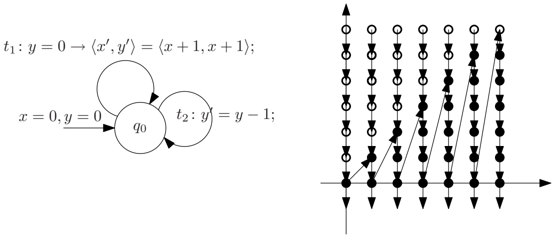

$t_1\colon y = 0 \to \langle x', y' \rangle = \langle x+1, x+1 \rangle;$
$x = 0, y = 0$
$q_0$
$t_2\colon y' = y - 1;$

**Fig. 2.** A two counters automata and its associated semantics

Our candidate relation to show termination is given by $t_1 \cup t_2^*$ which is computable using acceleration technique. It is routine to check that $T \subseteq t_1 \cup t_2^* \subseteq T^*$. Let us compute $Z_1$ which is given by $S \cap \widetilde{pre}[t_1 \cup t_2^*](S)$.

$$\begin{aligned} S \cap \widetilde{pre}[t_1 \cup t_2^*](S) &= && \text{def. } \widetilde{pre} \\ S \cap \widetilde{pre}[t_1](S) \cap \widetilde{pre}[t_2^*](S) &= && S \cap \widetilde{pre}[t_1](S) = S \\ \widetilde{pre}[t_2^*](S) &= && \\ \mathit{gfp}^{\subseteq} \lambda X.\, S \cap \widetilde{pre}[t_2](X) & && \end{aligned}$$

The latter fixpoint evaluates to $\{(x, y) \mid y \leqslant x\}$ and so the new abstract domain $A_1$ is such that $\gamma(A_1) = \mathcal{M}(\gamma(A_0) \cup Z_1)$. At iteration 1, we find at line 3 that $\gamma(\mathcal{R}_1) = Z_1$, hence that the test of line 4 succeeds since there is no outgoing transition of $Z_1$ (see Fig. 2), and finally that Alg. 1 terminates with the right answer.

It is worth pointing that the forward abstract semantics is conclusive. However algorithms using acceleration techniques to compute the forward concrete semantics do not terminate. Basically acceleration techniques identify regular expressions over the transition alphabet and then compute underapproximation of the transitive closure of the transition relation. For the automaton of Fig. 2 acceleration techniques fail because there is no finite regular expression that describes all the possible executions of the counter automaton. ■

The rest of this section is devoted to establish some termination properties of the enhanced algorithm. In fact, as we said in Rem. 1 our correctness proofs remains valid for the enhancement. Thus below we focus on termination properties.

By definition of $R$ it is routine to check that

$$\lambda X.\, \widetilde{pre}[T^*](X) \mathrel{\dot{\subseteq}} \lambda X.\, \widetilde{pre}[R](X) \mathrel{\dot{\subseteq}} \lambda X.\, \widetilde{pre}[T](X) \ . \tag{1}$$

**Proposition 8.** *Let $R_2$ such that $T \subseteq R_2 \subseteq T^*$ and $\mathit{gfp}^{\subseteq} \lambda X.\, S \cap \widetilde{pre}[R_2](X)$ stabilizes after a finite number of step, then Alg. 1 when using any $R_1$ such that $R_2 \subseteq R_1 \subseteq T^*$ at line 9 terminates as well.*

*Remark 2.* In Alg. 1, the fixpoint $\mathcal{R}_i$ and $\mathcal{S}_i$ have their iterated function given by $\lambda X.\, \alpha_i(I \cup \mathit{post}(\gamma(X)) \cap Z_i)$ and $\lambda X.\, \alpha_i(\gamma(\mathcal{R}_i) \cap \widetilde{pre}(\gamma(X)))$, respectively. For various reasons we may be constrained to use a less precise approximations $f, g$ that is to say $\lambda X.\, \alpha_i(I \cup \mathit{post}(\gamma(X)) \cap Z_i) \mathrel{\dot{\subseteq}} \lambda X.\, f(X)$ and $\lambda X.\, \alpha_i(\gamma(\mathcal{R}_i) \cap \widetilde{pre}(\gamma(X))) \mathrel{\dot{\subseteq}} \lambda X.\, g(X)$. In this context provided the additional mild requirements $\lambda X.\, f(X) \mathrel{\dot{\subseteq}} \lambda X.\, Z_i$ and $\lambda X.\, g(X) \mathrel{\dot{\subseteq}} \lambda X.\, \gamma(\mathcal{R}_i)$ hold we find that all the results of Sect. 3.1, Sect. 3.2 and Sect. 3.3 remain valid.

# 4 Relationships with Other Approaches

## 4.1 Counterexample Guided Abstraction Refinement

We first recall here the main ingredients of the CEGAR approach [21, §4.2]. Given a transition system $\mathcal{T} = (C, T, I)$, called the *concrete transition system*, and a partition of $C$ into a finite number of equivalence classes $\mathcal{C} = \{C_1, \ldots C_k\}$, the abstract transition system is a transition system $\mathcal{T}^\alpha = (C^\alpha, T^\alpha, I^\alpha)$ where:

- $C^\alpha = \mathcal{C}$, i.e. abstract states are the equivalence classes;
- $T^\alpha = \{(C_i, C_j) \mid \exists c \in C_i, c' \in C_j : (c, c') \in T\}$, i.e. there is a transition from an equivalence class $C_i$ to an equivalence class $C_j$ whenever there is a state of $C_i$ which has a successor in $C_j$ by the transition relation;
- $I^\alpha = \{C_i \in \mathcal{C} \mid C_i \cap I \neq \emptyset\}$, i.e. a class is initial whenever it contains an initial state.

A path in the abstract transition system is a finite sequence of abstract states related by $T^\alpha$ that starts in an initial state. An abstract state $C_i$ is reachable if there exists a path in $\mathcal{T}^\alpha$ that ends in $C_i$. The set of states within the equivalence classes that are reachable in the abstract transition system, is an overapproximation of the reachable states in the concrete transition system.

An abstract counterexample to $S \subseteq C$ is a path $C_{i_1}, C_{i_2}, \ldots, C_{i_n}$ in the abstract transition system such that $C_{i_n} \not\subseteq S$. An abstract counterexample is *spurious* if it does not match a concrete path in $\mathcal{T}$. We define this formally as follows. To an abstract counterexample $C_{i_1}, \ldots, C_{i_n}$, we associate a sequence $t_1, t_2, \ldots, t_{n-1}$ of subsets of $T$ (the transition relation of $\mathcal{T}$) such that $t_j = T \cap (C_{i_j} \times C_{i_{j+1}})$ (the projection of $T$ on successive classes).

An abstract counterexample is an *error trace*, only if $I \not\subseteq \widetilde{pre}[t_1 \circ \ldots \circ t_{n-1}](S)$ (by monotonicity we have $I \not\subseteq \widetilde{pre}[T^*](S)$), otherwise it is called *spurious* and, so $I \subseteq \widetilde{pre}[t_1 \circ \ldots \circ t_{n-1}](S)$. Eliminating a spurious counterexample is done by splitting a class $C_j$ where $1 \leqslant j \leqslant n$. The class $C_j$ contains *bad states* (written $\mathrm{bad}$) that can reach $\neg S$ but which are not reachable from $C_{j-1}$; or which are not initial if $j = 0$. Accordingly the class $C_j$ split in $C_j \cap \mathrm{bad}$ and $C_j \cap \neg\mathrm{bad}$. From the above definition, we can deduce that $\mathrm{bad} = pre[t_j \circ \ldots \circ t_{n-1}](\neg S)$, hence that $\neg\mathrm{bad} = \neg \circ pre[t_j \circ \ldots \circ t_{n-1}] \circ \neg(S)$, and, finally that $\neg\mathrm{bad} = \widetilde{pre}[t_j \circ \ldots \circ t_{n-1}](S)$. Hence the splitting of $C_j$ is given by $C_j \cap \widetilde{pre}[t_j \circ \ldots \circ t_{n-1}](S)$ and $C_j \cap \neg \circ \widetilde{pre}[t_j \circ \ldots \circ t_{n-1}](S)$. When the spurious counterexample has been removed, by splitting an equivalence class, a new abstract transition system, based on the refined partition, is considered and the method is iterated.

CEGAR approach concludes when it either finds an error trace (identifying a negative instance of the fixpoint checking problem) or when it does not find any new abstract counterexample (identifying a positive instance of the fixpoint problem).

We now relate the abstract model used by CEGAR with the abstract interpretation of the system. The initial abstract domain $A_0$, that our algorithm uses, is such that for all equivalence classes $C_i$ in the initial partition used by the CEGAR algorithm, there exists an abstract value $a \in A_0$ such that $\gamma(a) = C_i$.

**Lemma 4.** *Assume that CEGAR terminates on a positive instance of the fixpoint checking problem. So CEGAR produced a finite set $\{w_i\}_{i \in I}$ of counterexamples such that the following holds:*

$$\exists A \in \gamma(A_0) \colon I \subseteq \underbrace{gfp^{\subseteq} \lambda X.\, A \cap \widetilde{pre}(X)}_{V} \subseteq S \,\&\, V = A \cap \bigcap_{i \in I} \widetilde{pre}[w_i](S) \enspace .$$

We need one more auxiliary result before presenting Th. 1.

**Proposition 9.** *In Alg. 1, $\forall k \in \mathbb{N}$ if $post^*(\gamma(\mathcal{R}_k)) \subseteq S$ then $post(\gamma(\mathcal{R}_k)) \subseteq Z_k$.*

**Theorem 1.** *Assume a positive instance of the fixpoint checking problem, if CEGAR terminates so does Alg. 1.*

*Proof.* Let $k$ be the size of the longest $w_i$ for $i \in I$. Lem. 3 shows that $\gamma(\mathcal{R}_{k+1})$ is an underapproximation of the states that cannot escape $S$ in less than $k$ steps. Formally, we have $\gamma(\mathcal{R}_{k+1}) \subseteq \bigcap_{j=0}^{k} \widetilde{pre}[T^j](S)$. This implies that

$$\gamma(\mathcal{R}_{k+1}) \subseteq \bigcap_{i \in I} \widetilde{pre}[w_i](S) \ . \tag{2}$$

Our next step will be to show that $post[T^*](\gamma(\mathcal{R}_{k+1})) \subseteq S$ which intuitively says that $\gamma(\mathcal{R}_{k+1})$ cannot escape $S$. First, note that if $\gamma(\mathcal{R}_{k+1})$ can escape from $S$ then it cannot be with the counterexamples produced by CEGAR since $\gamma(\mathcal{R}_{k+1}) \subseteq \bigcap_{i \in I} \widetilde{pre}[w_i](S)$ which is equivalent to $\bigcup_{i \in I} post[w_i](\gamma(\mathcal{R}_{k+1})) \subseteq S$ by $\underset{post}{\overset{\widetilde{pre}}{\leftrightarrows}}$. Let $A$ be defined as in Lem. 4. Our proof falls into two parts:

1. $\gamma(\mathcal{R}_{k+1}) \cap A$ cannot escape from $S$, i.e. $post[T^*](\gamma(\mathcal{R}_{k+1}) \cap A) \subseteq S$, as shown as follows. From (2), we know that $\gamma(\mathcal{R}_{k+1}) \subseteq \bigcap_{i \in I} \widetilde{pre}[w_i](S)$, and by definition of $V$, we have that $\gamma(\mathcal{R}_{k+1}) \cap A \subseteq V$. As $V$ is inductive for $post$ and $V \subseteq S$, we conclude that $post[T^*](\gamma(\mathcal{R}_{k+1}) \cap A) \subseteq S$.
2. $\gamma(\mathcal{R}_{k+1}) \cap \neg A$ cannot escape from $S$. For that, we show that $\gamma(\mathcal{R}_{k+1}) \cap \neg A = \emptyset$. Prop. 1 and definition of $A$ show that $I \subseteq \gamma(\mathcal{R}_{k+1}) \cap A$ and so $\gamma(\mathcal{R}_{k+1}) \cap A \neq \emptyset$. We also know that in any state $s \in \gamma(\mathcal{R}_{k+1}) \cap A$ for $post[T^*](\{s\}) \cap \neg A \neq \emptyset$ to hold $s$ has to be such that $s \notin \bigcap_{i \in I} \widetilde{pre}[w_i](S)$. However since $\gamma(\mathcal{R}_{k+1}) \subseteq \bigcap_{i \in I} \widetilde{pre}[w_i](S)$ and since $\mathcal{R}_{k+1}$ is given by $lfp^{\subseteq} \lambda X.\, \alpha_{k+1}(I \cup post(\gamma(X)) \cap Z_{k+1})$ over $A_{k+1}$ (with $\gamma(A_{k+1}) \supseteq \gamma(A_0)$) we find that $\gamma(\mathcal{R}_{k+1}) \cap \neg A = \emptyset$. It follows that $post[T^*](\gamma(\mathcal{R}_{k+1})) \subseteq S$.

We conclude from Prop. 9 that $post(\gamma(\mathcal{R}_{k+1})) \subseteq Z_{k+1}$, hence that the test of line 4 succeeds by $\alpha_{k+1}$ monotonicity and $I \subseteq \gamma(\mathcal{R}_{k+1})$, and finally that Alg. 1 terminates. □

If we consider the converse result, namely that CEGAR terminates if Alg. 1 terminates we find that this does not hold for the enhanced algorithm as shown in Ex. 2.

### 4.2 Predicate Abstraction Versus Moore Closed Abstract Domains

Below we prove that Alg. 1 does not take any advantage maintaining a Boolean closed abstract domain instead of a Moore closed one: Moore closure is as strong as Boolean closure.

The following Lemma shows that every "interesting" value added by the Boolean closure is added by the Moore closure as well. By extension we obtain that (see Th. 2) if Alg. 1 extended with the Boolean closure terminates then Alg. 1 terminates. Our result holds basically because both $\mathcal{R}_i$ and $\mathcal{S}_i$ are such that $\gamma(\mathcal{R}_i) \subseteq Z_i$ and $\gamma(\mathcal{S}_i) \subseteq Z_i$ by Lem. 2.

**Lemma 5.** *Let $A$ be a finite subset of $L$ such that $\mathcal{B}(A) = A$ and let $Z_0, Z_1, \ldots, Z_k$ be elements of $L$ such that $Z_k \subseteq \cdots \subseteq Z_1 \subseteq Z_0$. Given $e \in \mathcal{B}(A \cup \{Z_0, Z_1, \ldots, Z_k\})$ such that $e \subseteq Z_k$ we have $e \in \mathcal{M}(A \cup \{Z_0, Z_1, \ldots, Z_k\})$.*

**Theorem 2.** *Provided $\mathcal{B}(\gamma(A_0)) = \gamma(A_0)$, if Alg. 1 with the Moore closure (viz. $\mathcal{M}$) replaced by the Boolean closure (viz $\mathcal{B}$) terminates then Alg. 1 terminates as well.*

In the context of predicate abstraction, there is no polynomial algorithm to compute the best approximation. In fact the result of applying $\alpha$ to value $V$ is given by the strongest Boolean combination of predicates approximating $V$. Moreover the computation of the best approximation is required at each iterate of each fixpoint computation. So in the worst case the time to compute a fixpoint is given by the height of the abstract lattice times an exponential in the number of predicates. It is generally admitted that this cost is not affordable and this is why approximations in time linear in the number of predicates are preferred instead. For our algorithm the situation is pretty much better: as shown in Lem. 5 we can compute the best approximation in time linear in the number of predicates. However we need the initial set of predicates to be Boolean closed.

## 5 Conclusion and Future Works

We have presented a new abstract fixpoint refinement algorithm for the fixpoint checking problem. Our systematic refinement uses the information computed so far which is given by two fixpoints computed in the abstract domain. As a future work, we can consider two variants of this algorithm. First, the dual algorithm for the inverted transition system $T^{-1}$ can be used to discover necessary correct termination conditions. A second dual algorithm where we use the inverted inclusion order $\supseteq$ on states leading to underapproximation of fixpoints. In this settings the *lfp* allows to conclude on negative instances and the *gfp* on positive instances. Also the refinement step uses the $post$ predicate transformer instead of $\widetilde{pre}$. Finally we will consider more complicated properties like properties defined by nested fixpoint expressions.

## References

1. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL'77: Proc. 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages, pp. 238–252. ACM Press, New York (1977)
2. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with blast. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)
3. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in c. In: ICSE '03: Proc. 25th International Conference on Software Engineering, pp. 385–395. IEEE Computer Society Press, Los Alamitos (2003)
4. Ball, T., Rajamani, S.K.: The slam project: debugging system software via static analysis. In: POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 1–3. ACM Press, New York (2002)

5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
6. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL '02: Proc. 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp. 58–70. ACM Press, New York (2002)
7. Cousot, P.: Méthodes Itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French). Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble (1978)
8. Massé, D.: Combining forward and backward analyses of temporal properties. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 103–116. Springer, Heidelberg (2001)
9. Giacobazzi, R., Quintarelli, E.: Incompleteness, counterexamples and refinements in abstract model-checking. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 356–373. Springer, Heidelberg (2001)
10. Ball, T., Podelski, A., Rajamani, S.K.: Relative completeness of abstraction refinement for software model checking. In: Katoen, J.-P., Stevens, P. (eds.) ETAPS 2002 and TACAS 2002. LNCS, vol. 2280, pp. 158–172. Springer, Heidelberg (2002)
11. Lakhnech, Y., Bensalem, S., Berezin, S., Owre, S.: Incremental verification by abstraction. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 98–112. Springer, Heidelberg (2001)
12. Cousot, P., Cousot, R.: Refining model checking by abstract interpretation. Automated Software Engineering 6(1), 69–95 (1999)
13. Cousot, P.: Partial completeness of abstract fixpoint checking, invited paper. In: Choueiry, B.Y., Walsh, T. (eds.) SARA 2000. LNCS (LNAI), vol. 1864, pp. 1–25. Springer, Heidelberg (2000)
14. Cousot, P.: Semantic foundations of program analysis. In: Muchnick, S.S, Jones, N.D (eds.) Program Flow Analysis: Theory and Applications, pp. 303–342. Prentice-Hall, Englewood Cliffs (1981)
15. Esparza, J., Ganty, P., Schwoon, S.: Locality-based abstractions. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 118–134. Springer, Heidelberg (2005)
16. Ganty, P., Raskin, J.-F., Van Begin, L.: A complete abstract interpretation framework for coverability properties of WSTS. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 49–64. Springer, Heidelberg (2005)
17. Ganty, P., Raskin, J.-F., Van Begin, L.: From many places to few: Automatic abstraction refinement for Petri Nets. In: ATPN'07. LNCS, Springer, Heidelberg (2007)
18. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: Proc.1th Annual IEEE Symp. on Logic in Computer Science (LICS), pp. 313–321. IEEE Computer Society Press, Los Alamitos (1996)
19. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science 256(1-2), 1–2 (1997)
20. Boigelot, B.: On iterating linear transformations over recognizable sets of integers. Theoretical Computer Science 309(2), 413–468 (2003)
21. Dams, D.: Comparing abstraction refinement algorithms. Electr. Notes Theor. Comput. Sci 89(3) (2003)

# Guided Static Analysis*

Denis Gopan[1] and Thomas Reps[1,2]

[1] University of Wisconsin
[2] GrammaTech, Inc.
{gopan,reps}@cs.wisc.edu

**Abstract.** In static analysis, the semantics of the program is expressed as a set of equations. The equations are solved iteratively over some abstract domain. If the abstract domain is distributive and satisfies the ascending-chain condition, an iterative technique yields the most precise solution for the equations. However, if the above properties are not satisfied, the solution obtained is typically imprecise. Moreover, due to the properties of widening operators, the precision loss is sensitive to the order in which the state-space is explored.

In this paper, we introduce *guided static analysis*, a framework for controlling the exploration of the state-space of a program. The framework guides the state-space exploration by applying standard static-analysis techniques to a sequence of modified versions of the analyzed program. As such, the framework does not require any modifications to existing analysis techniques, and thus can be easily integrated into existing static-analysis tools.

We present two instantiations of the framework, which improve the precision of widening in (i) loops with multiple phases and (ii) loops in which the transformation performed on each iteration is chosen non-deterministically.

## 1 Introduction

The goal of static analysis is, given a program and a set of initial states, to compute the set of states that arise during the execution of the program. Due to general undecidability of this problem, the sets of program states are typically over-approximated by families of sets that both are decidable and can be effectively manipulated by a computer. Such families are referred to as abstractions or abstract domains. In static analysis, the semantics of the program is cast as a set of equations, which are solved iteratively over a chosen abstract domain. If the abstract domain possesses certain algebraic properties, namely, if the abstract transformers for the domain are monotonic and distribute over join, and if the domain does not contain infinite strictly-increasing chains, then simple iterative techniques yield the least fix-point for the set of equations.

However, many useful existing abstract domains, especially those for modeling numeric properties, do not possess the above algebraic properties. As a result, standard iterative techniques (augmented with widening, to ensure analysis convergence) tend to lose precision. The precision is lost both due to overly-conservative invariant guesses

* Supported by ONR under grant N00014-01-1-0796 and by NSF under grants CCF-0540955 and CCF-0524051.

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 349–365, 2007.

made by widening, and due to joining together the sets of reachable states along multiple paths. In previous work [11], we showed that the loss of precision can sometimes be avoided by forcing the analysis to explore the state space of the program in a certain order. In particular, we showed that the precision of widening in loops with multiple phases can be improved if the analysis has a chance to precisely characterize the behavior of each phase before having to account for the behavior of subsequent phases.

In this paper, we introduce *guided static analysis*, a general framework for guiding state-space exploration. The framework controls state-space exploration by applying standard static-analysis techniques to a sequence of *program restrictions*, which are modified versions of the analyzed program. The result of each analysis run is used to derive the next program restriction in the sequence, and also serves as an approximation of a set of initial states for the next analysis run. Note that existing static-analysis techniques are utilized "as is", making it easy to integrate the framework into existing tools. The framework is instantiated by specifying a procedure for deriving program restrictions.

We present two instantiations of the framework. The first instantiation improves the precision of widening in loops that have multiple phases. This instantiation generalizes the lookahead-widening technique [11]. It operates by generating program restrictions that incorporate individual loop phases. Also, it lifts the limitations of lookahead widening, such as the restrictions imposed on the iteration strategy and on the length of the descending-iteration sequence.

The second instantiation addresses the precision of widening in loops where the behavior of each iteration is chosen non-deterministically. Such loops naturally occur in the realm of synchronous systems [13,10] and can occur in imperative programs if some condition within a loop is abstracted away. This instantiation derives a sequence of program restrictions, each of which enables a single iteration behavior and disables all of the others. At the end, to make the analysis sound, a program restriction with all behaviors enabled is analyzed. This strategy allows the analysis to characterize each behavior in isolation, thereby obtaining more precise results.

In non-distributive domains, the join operation loses precision. To keep the analysis precise, many techniques propagate sets of abstract values instead of individual values. Various heuristics are used to keep the cardinalities of propagated sets manageable. The main question that these heuristics address is which abstract elements should be joined and which must be kept separate. Guided static analysis is comprised of a sequence of phases, where each phase derives and analyzes a program restriction. The phase boundaries are natural points for separating abstract values: that is, within each phase the analysis may propagate a single abstract value; however, the results of different phases need not be joined together, but may be kept as a set, thus yielding a more precise overall result. In §5, we show how to extend the framework to take advantage of such disjunctive partitioning.

We implemented a prototype of guided static analysis with both of the instantiations, and applied them to a set of small programs that have appeared in recent literature on widening. The first instantiation and its disjunctive extension were used to analyze the benchmarks from [11]. The results were compared against those produced by lookahead widening. As expected, the results obtained by the instantiation were similar to the ones in [11]. However, the results obtained with the disjunctive extension were much more precise. The second instantiation was used to analyze the examples from [10]. The obtained results were similar to the ones in [10]. However, we believe that our approach is conceptually simpler because it does not rely on acceleration techniques.

**Contributions.** In this paper, we make the following contributions:

- we introduce a general framework for guiding state-space exploration; the framework utilizes existing static-analysis techniques, which makes it easy to integrate into existing tools.
- we present two instantiations of the framework, which improve the precision of widening in (i) loops that have multiple phases; (ii) loops in which the transformations performed on each iteration are selected non-deterministically.
- we describe a disjunctive extension of the framework.
- we present an experimental evaluation of our techniques.

**Paper organization.** §2 defines the basic concepts used in the rest of the paper; §3 introduces the framework; §4 describes the two instantiations of the framework; §5 presents the disjunctive extension of the framework; §6 gives the experimental results; §7 reviews related work.

## 2 Preliminaries

We assume that a program is specified by a *control flow graph (CFG)* $G = (V, E)$, where $V$ is a set of program locations, and $E \sqsubseteq V \times V$ is a set of edges that represent the flow of control. A *program state* assigns a value to every variable in the program. We will use $\Sigma$ to denote the set of all possible program states. The function $\Pi_G : E \to (\Sigma \to \Sigma)$ assigns to each edge in the CFG the concrete semantics of the corresponding program statement. The semantics of individual statements is trivially extended to operate on sets of states, i.e., $\Pi_G(e)(S) = \{\Pi_G(e)(s) \mid s \in S\}$, where $e \in E$ and $S \subseteq \Sigma$.

Let $\Theta_0 : V \to \wp(\Sigma)$ denote a mapping from program locations to sets of states. The sets of program states that are *reachable* at each program location from the states in $\Theta_0$ are given by the least map $\Theta_\star : V \to \wp(\Sigma)$ that satisfies the following set of equations:

$$\Theta_\star(v) \supseteq \Theta_0(v), \quad \text{and} \quad \Theta_\star(v) = \bigcup_{\langle u,v\rangle \in E} \Pi_G(\langle u, v\rangle)(\Theta_\star(u)), \text{ for all } v \in V$$

The problem of computing sets of reachable states is, in general, undecidable.

**Static Analysis.** Static analysis sidesteps undecidability by using abstraction: sets of program states are approximated by elements of some abstract domain $\mathbb{D} = \langle D, \alpha, \gamma, \sqsubseteq, \top, \bot, \sqcup\rangle$, where $\alpha : \wp(\Sigma) \to D$ constructs an approximation for a set of states, $\gamma : D \to \wp(\Sigma)$ gives meaning to domain elements, $\sqsubseteq$ is a partial order on $D$, $\top$ and $\bot$ are, respectively, the least and the greatest elements of $D$, and $\sqcup$ is the least upper bound operator. The function $\Pi_G^\sharp : E \to (D \to D)$ gives the abstract semantics of individual program statements.

To refer to abstract states at multiple program locations, we define *abstract-state maps* $\Theta^\sharp : V \rightarrow D$. The operations $\alpha$, $\gamma$, $\sqsubseteq$, and $\sqcup$ for $\Theta^\sharp$ are point-wise extensions of the corresponding operations for $\mathbb{D}$.

A static analysis computes an approximation for the set of states that are reachable from an approximation of the set of initial states according to the abstract semantics of the program. In the rest of the paper, we view static analysis as a black box, denoted by $\Omega$, with the following interface: $\Theta^\sharp_\star = \Omega(\Pi^\sharp_G, \Theta^\sharp_0)$, where $\Theta^\sharp_0 = \alpha(\Theta_0)$ is the initial abstract-state map, and $\Theta^\sharp_\star$ is an abstract-state map that satisfies the following property:

$$\forall v \in V : \left[ \Theta^\sharp_0(v) \sqcup \bigsqcup_{\langle u,v\rangle \in E} \Pi^\sharp_G(\langle u,v\rangle)(\Theta^\sharp_\star(u)) \right] \sqsubseteq \Theta^\sharp_\star(v).$$

## 3 Guided Static Analysis

A *guided static analysis* framework provides control over the exploration of the state space. Instead of constructing a new analysis by means of designing a new abstract domain or imposing restrictions on existing analyses (e.g., by fixing an iteration strategy), the framework relies on existing static analyses “as is”. Instead, state-space exploration is guided by modifying the analyzed program to restrict some of its behaviors; multiple analysis runs are performed to explore all of the program’s behaviors.

The framework is parametrized with a procedure for deriving such program restrictions. The analysis proceeds as follows: the initial abstract-state map, $\Theta^\sharp_0$, is used to derive the first program restriction; standard static analysis is applied to that program restriction to compute $\Theta^\sharp_1$, which approximates a set of program states reachable from $\Theta^\sharp_0$. Then, $\Theta^\sharp_1$ is used to derive the second program restriction, which is in turn analyzed by a standard analysis to compute $\Theta^\sharp_2$. This process is repeated until the $i$-th derived restriction is equivalent to the original program; the final answer is $\Theta^\sharp_i$.

We use the program in Fig. 1(a) to illustrate guided static analysis framework. The loop in the program has two explicit phases: during the first fifty iterations both variable $x$ and variable $y$ are incremented; during the next fifty iterations variable $x$ is incremented and variable $y$ is decremented. The loop exits when the value of the variable $y$ falls below $0$. This program is a challenge for standard widening/narrowing-based numeric analyses because the application of the widening operator over-approximates the behavior of the first phase and initiates the analysis of the second phase with overly-conservative initial assumptions. As a result, polyhedra-based standard numeric analysis concludes that at the program point $n_1$ the relationship between the values of $x$ and $y$ is $0 \leq y \leq x$, and at the program point $n_x$, $y = -1$ and $x \geq 50$. This is imprecise compared to the true sets of states at those program points (Figs. 1(c) and 1(d)).

Guided static analysis, when applied to the program in Fig. 1(a) consecutively derives three program restrictions shown in Fig. 2: (a) consists to the first phase of the program; (b) incorporates both phases, but excludes the edge that leads out of the loop; (c) includes the entire program. Each restriction is formed by substituting abstract transformers associated with certain edges in the control flow graph with more restrictive transformers (in this case, with $\bot$, which is equivalent to removing the edge from the graph). We defer the description of the procedure for deriving these restrictions to §4.1.

```
x = 0;
y = 0;

while(true)
{
    if(x <= 50) y++;
      else y--;

    if(y < 0) break;

    x++;
}
```

(a)

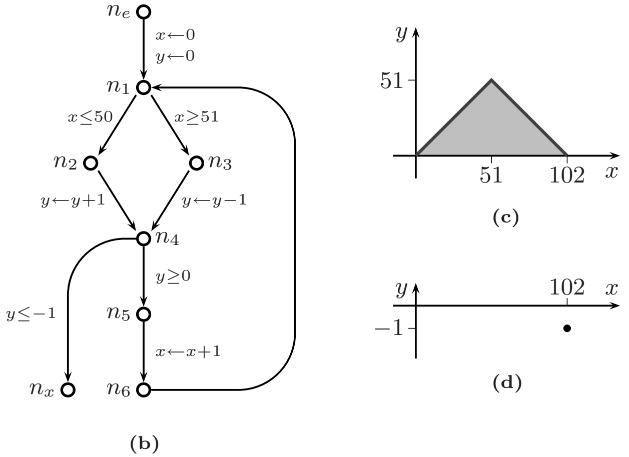


**Fig. 1.** Running example: (a) a loop with non-regular behavior; (b) control-flow graph for the program in (a); (c) the set of program states at $n_1$: the points with integer coordinates that lie on the dark upside-down "v" form the precise set of concrete states; the gray triangle gives the best approximation of that set in the polyhedral domain; (d) the single program state that reaches $n_x$

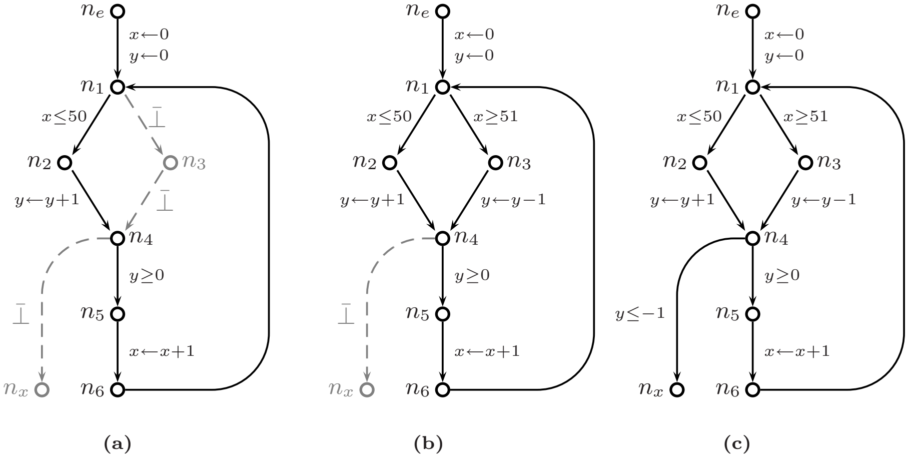
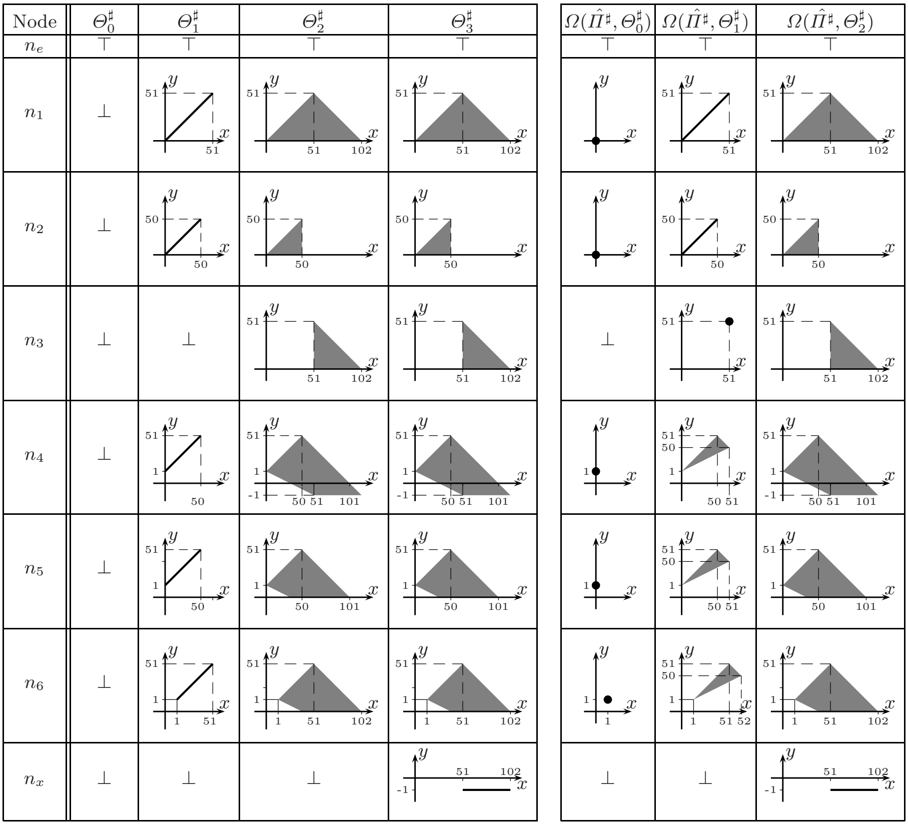


**Fig. 2.** Program restrictions for the program in Fig. 1: the unreachable portions of each CFG are shown in gray; (a) the first restriction corresponds to the first loop phase; (b) the second restriction consists of both loop phases, but not the loop-exit edge; (c) the third restriction incorporates the entire program

Fig. 3(a) illustrates the operation of guided static analysis. $\Theta_0^\sharp$ approximates the set of initial states of the program. The standard numeric analysis, when applied to the first restriction (Fig. 2(a)), yields the abstract-state map $\Theta_1^\sharp$, i.e., $\Theta_1^\sharp = \Omega(\Pi_1^\sharp, \Theta_0^\sharp)$. Note, that the invariant for the first loop phase ($0 \leq x = y \leq 51$) is captured precisely.



(a) (b)

**Fig. 3.** Guided static analysis results for the program in Fig. 1(a); **(a)** the sequence of abstract states that are computed by analyzing the program restrictions shown in Fig. 2; $\Theta_3^\sharp$ is the overall result of the analysis; **(b)** the abstract states that are obtained by analyzing the acyclic version of the program, which are used to construct the program restrictions in Fig. 2 (see §4.1)

Similarly, $\Theta_2^\sharp$ is computed as $\Omega(\Pi_2^\sharp, \Theta_1^\sharp)$, and $\Theta_3^\sharp$ is computed as $\Omega(\Pi_3^\sharp, \Theta_2^\sharp)$. Since the third restriction is equivalent to the program itself, the analysis stops, yielding $\Theta_3^\sharp$ as the overall result. Note that $\Theta_3^\sharp$ is more precise than the solution computed by the standard analysis: it precisely captures the loop invariant at program point $n_1$ and the upper bound for the value of $x$ at node $n_x$. In fact, $\Theta_3^\sharp$ corresponds to the least fix-point for the program in Fig. 1(a) in the polyhedral domain.

### 3.1 Formal Description

We start by extending the partial order of the abstract domain to abstract transformers and to entire programs. The order is extended in a straightforward fashion.

**Definition 1.** *Let $f, g : D \to D$ be two abstract transformers, let $G = (V, E)$ be a control-flow graph, and let $\Pi_1^\sharp, \Pi_2^\sharp : E \to (D \to D)$ be two programs specified over $G$. Then we say that (i) $f \bar{\sqsubseteq} g$ iff $\forall d \in D : f(d) \sqsubseteq g(d)$; and (ii) $\Pi_1^\sharp \dot{\sqsubseteq} \Pi_2^\sharp$ iff $\forall e \in E : \Pi_1^\sharp(e) \bar{\sqsubseteq} \Pi_2^\sharp(e)$.*

A program restriction is a version of a program $\Pi^\sharp$ in which some abstract transformers under-approximate ($\bar{\sqsubseteq}$) those of $\Pi$. The aim is to make a standard analysis (applied to the restriction) explore only a subset of reachable states of the original program. Note, however, that, if widening is used by the analyzer, there are no guarantees that the explored state space would be smaller (because widening is, in general, not monotonic).

**Definition 2 (Program Restriction).** *Let $G = (V, E)$ be a control-flow graph, and $\Pi^\sharp : E \to (D \to D)$ be a program specified over $G$. We say that $\Pi_r^\sharp : E \to (D \to D)$ is a* restriction *of $\Pi^\sharp$ if $\Pi_r^\sharp \dot{\sqsubseteq} \Pi^\sharp$*

To formalize guided static analysis, we need a notion of a *program transformer*: that is, a procedure $\Lambda$ that, given a program and an abstract state, derives a corresponding program restriction. We allow a program transformer to maintain internal states, the set of which will be denoted $\mathbb{I}$. We assume that the set $\mathbb{I}$ is defined as part of $\Lambda$.

**Definition 3 (Program transformer).** *Let $\Pi^\sharp$ be a program, let $\Theta^\sharp : V \to D$ be an arbitrary abstract-state map, and let $I \in \mathbb{I}$ be an internal state of the program transformer. A* program transformer*, $\Lambda$, computes a restriction of $\Pi^\sharp$ with respect to $\Theta^\sharp$, and modifies its internal state, i.e.:*

$$\Lambda(\Pi^\sharp, I, \Theta^\sharp) = (\Pi_r^\sharp, I_r), \quad \textit{where } \Pi_r^\sharp \dot{\sqsubseteq} \Pi^\sharp \textit{ and } I_r \in \mathbb{I}.$$

To ensure the soundness and the convergence of the analysis, we require that the program transformer possess the following property: the sequence of program restrictions generated by a non-decreasing chain of abstract states must converge to the original program in finitely many steps.

**Definition 4 (Chain Property).** *Let $(\Theta_i^\sharp)$ be a non-decreasing chain, s.t., $\Theta_0^\sharp \sqsubseteq \Theta_1^\sharp \sqsubseteq ... \sqsubseteq \Theta_k^\sharp \sqsubseteq ....$ Let $(\Pi_i^\sharp)$ be a sequence of program restrictions derived from $(\Theta_i^\sharp)$ as follows:*

$$(\Pi_{i+1}^\sharp, I_{i+1}) = \Lambda(\Pi^\sharp, I_i, \Theta_i^\sharp)$$

*where $I_0$ is the initial internal state for $\Lambda$. We say that $\Lambda$ satisfies the chain property if there exists a natural number $n$ such that $\Pi_i^\sharp = \Pi^\sharp$, for all $i \geq n$.*

The above property is not burdensome: any mechanism for generating program restrictions can be forced to satisfy the property by introducing a threshold and returning the original program after the threshold has been exceeded.

**Definition 5 (Guided Static Analysis).** *Let $\Pi^\sharp$ be a program, and let $\Theta_0^\sharp$ be an initial abstract-state map. Also, let $I_0$ be an initial internal state for the program transformer $\Lambda$. Guided static analysis performs the following sequence of iterations:*

$$\Theta_{i+1}^\sharp = \Omega(\Pi_{i+1}^\sharp, \Theta_i^\sharp), \ \textit{where} \ (\Pi_{i+1}^\sharp, I_{i+1}) = \Lambda(\Pi^\sharp, I_i, \Theta_i^\sharp),$$

*until $\Pi_{i+1}^\sharp = \Pi^\sharp$. The analysis result is $\Theta_\star^\sharp = \Theta_{i+1}^\sharp = \Omega(\Pi_{i+1}^\sharp, \Theta_i^\sharp) = \Omega(\Pi^\sharp, \Theta_i^\sharp)$.*

Let us show that if the program transformer satisfies the chain property, the above analysis is sound and converges in a finite number of steps. Both arguments are trivial:

**Soundness.** Let $\Pi_a^\sharp$ be an arbitrary program and let $\Theta_a^\sharp$ be an arbitrary abstract-state map. Due to the soundness of $\Omega$, the following holds: $\Theta_a^\sharp \sqsubseteq \Omega(\Pi_a^\sharp, \Theta_a^\sharp)$. Now, let $(\Pi_i^\sharp)$ be a sequence of programs and let $(\Theta_i^\sharp)$ be a sequence of abstract-state maps computed according to the procedure in Defn. 5. Since each $\Theta_i^\sharp$ is computed as $\Omega(\Pi_i^\sharp, \Theta_{i-1}^\sharp)$, clearly, the following relationship holds: $\Theta_0^\sharp \sqsubseteq \Theta_1^\sharp \sqsubseteq ... \sqsubseteq \Theta_k^\sharp \sqsubseteq ....$

Since $\Lambda$ satisfies the chain property, there exists a number $n$ such that $\Pi_i^\sharp = \Pi^\sharp$ for all $i \geq n$. The result of the analysis is computed as

$$\Theta_\star^\sharp = \Theta_n^\sharp = \Omega(\Pi_n^\sharp, \Theta_{n-1}^\sharp) = \Omega(\Pi^\sharp, \Theta_{n-1}^\sharp)$$

and, since $\Theta_0^\sharp \sqsubseteq \Theta_{n-1}^\sharp$ (i.e., the $n$-th iteration of the analysis computes a set of program states reachable from an over-approximation of the set of initial states, $\Theta_0^\sharp$), it follows that guided static analysis is sound.

**Convergence.** Convergence follows trivially from the above discussion: since $\Pi_n^\sharp = \Pi^\sharp$ for some finite number $n$, guided static analysis converges after $n$ iterations.

## 4 Framework Instantiations

The framework of guided static analysis is instantiated by supplying a suitable program transformer, $\Lambda$. This section presents two instantiations that are aimed at recovering precision lost due to the use of widening.

### 4.1 Widening in Loops with Multiple Phases

As was illustrated in §3, multiphase loops pose a challenge for standard analysis techniques. The problem is that standard techniques are not able to invoke narrowing after the completion of each phase to refine the analysis results for that phase. Instead, narrowing is invoked at the very end of the analysis when the accumulated precision loss is too great for precision to be recovered.

In previous work, we proposed a technique called *lookahead widening* that addressed this problem [11]. Lookahead widening propagated a pair of abstract values through the program: the first value was used to "lock" the analysis within the current loop phase; the second value computed the solution for the current phase and refined it with a narrowing sequence. When the second value converged, it was moved into the first value, thereby allowing the next loop phase to be considered. To make lookahead widening work in practice, certain restrictions were placed on the iteration strategy used by the analysis; also, the length of the descending-iteration sequence was limited to one. Furthermore, very short loop phases caused precision loss if the first value allowed the analysis to exit the current loop phase before the second value was able to converge.

In this section, we present an instantiation of the guided static analysis framework that generalizes lookahead widening and lifts the above restrictions and limitations. To instantiate the framework, we need to construct a program transformer, $\Lambda_{phase}$, that derives program restrictions that isolate individual loop phases (as shown in Fig. 2). Intuitively, given an abstract-state map, we would like to include into the generated restriction the edges that are immediately exercised by that abstract state, and exclude the edges that require several loop iterations to become active.

To define the program transformer, we again rely on the application of a standard static analysis to a modified version of the program. Let $\hat{\Pi}^\sharp$ denote the version of $\Pi^\sharp$ from which all backedges have been removed. Note that the program $\hat{\Pi}^\sharp$ is acyclic and thus can be analyzed efficiently and precisely. The program transformer $\Lambda_{phase}(\Pi^\sharp, \Theta^\sharp)$ is defined as follows (no internal states are maintained, so we omit them for brevity):

$$\Pi_r^\sharp(\langle u, v\rangle) = \begin{cases} \Pi^\sharp(\langle u, v\rangle) & \text{if } \Pi^\sharp(\langle u, v\rangle)(\Omega(\hat{\Pi}^\sharp, \Theta^\sharp)(u)) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

In practice, we first analyze the acyclic version of the program: $\hat{\Theta}^\sharp = \Omega(\hat{\Pi}^\sharp, \Theta^\sharp)$. Then, for each edge $\langle u, v\rangle \in E$, we check whether that edge should be included in the program restriction: if the edge is active (that is, if $\Pi^\sharp(\langle u, v\rangle)(\hat{\Theta}^\sharp(u))$ yields a non-bottom value), then the edge is included in the restriction; otherwise, it is omitted.

Fig. 3(b) illustrates this process for the program in Fig. 1(a). $\hat{\Pi}^\sharp$ is constructed by removing the edge $\langle n_6, n_1\rangle$ from the program. The first column in Fig. 3(b) shows the result of analyzing $\hat{\Pi}^\sharp$ with $\Theta_0^\sharp$ used as the initial abstract-state map. The transformers associated with the edges $\langle n_1, n_3\rangle$, $\langle n_3, n_4\rangle$, and $\langle n_4, n_x\rangle$ yield $\bot$ when applied to the analysis results. Hence, these edges are excluded from the program restriction $\Pi_1^\sharp$ (see Fig. 2(a)). Similarly, the abstract-state map shown in the second column of Fig. 3(b) excludes the edge $\langle n_4, n_x\rangle$ from the restriction $\Pi_2^\sharp$. Finally, all of the edges are active with respect to the abstract-state map shown in the third column. Thus, the program restriction $\Pi_3^\sharp$ is equivalent to the original program.

Note that the program transformer $\Lambda_{phase}$, as defined above, does not satisfy the chain property from Defn. 4: arbitrary non-decreasing chains of abstract-state maps may not necessarily lead to the derivation of program restrictions that are equivalent to the original program. However, note that the process is bound to converge to some program restriction after a finite number of steps. To see this, note that each consecutive program restriction contains all of the edges included in the previously generated restrictions, and the overall number of edges in the program's CFG is finite. Thus, to satisfy the chain property, we make $\Lambda_{phase}$ return $\Pi^\sharp$ after convergence is detected.

### 4.2 Widening in Loops with Non-deterministically Chosen Behavior

Another challenge for standard analysis techniques is posed by loops in which the behavior of each iteration is chosen non-deterministically. Such loops often arise when modeling and analyzing synchronous systems [13,10], but they may also arise in the analysis of imperative programs when a condition of an if statement in the body of the loop is abstracted away (e.g., if variables used in the condition are not modeled by the analysis). These loops are problematic due to the following two reasons:

- the analysis may be forced to explore multiple iteration behaviors at the same time (e.g., simultaneously explore multiple arms of a non-deterministic conditional), making it hard for widening to predict the overall behavior of the loop accurately;

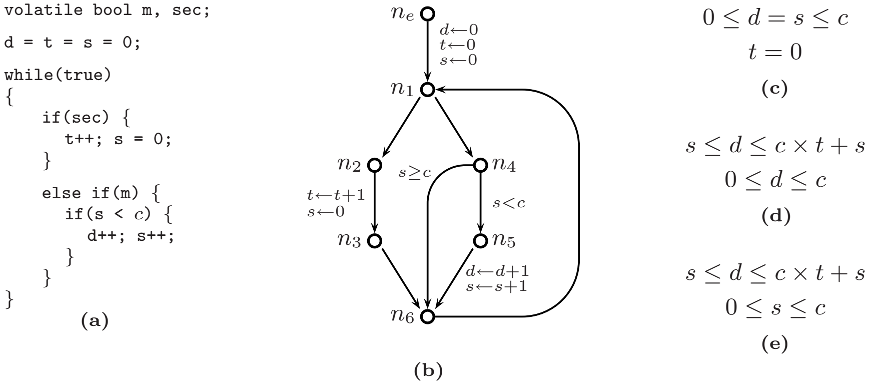
– narrowing is not effective in such loops: narrowing operates by filtering an overapproximation of loop behavior through the conditional statements in the body of the loop; in these loops, however, the relevant conditional statements are buried within the arms of a non-deterministic conditional, and the join operation at the point where the arms merge cancels the effect of such filtering.

Fig. 4(a) shows an example of such loop: the program models a speedometer with the assumption that the maximum speed is $c$ meters per second ($c > 0$ is an arbitrary integer constant) [10]. Variables $m$ and $sec$ model signals raised by a time sensor and a distance sensor, respectively. Signal $sec$ is raised every time a second elapses: in this case, the time variable $t$ is incremented and the speed variable $s$ is reset. Signal $m$ is raised every time a distance of one meter is traveled: in this case, both the distance variable $d$ and the speed variable $s$ are incremented. Fig. 4(b) shows the CFG for the program: the environment (i.e., the signals issued by the sensors) is modeled non-deterministically (node $n_1$). The invariant that we desire to obtain at node $n_1$ is $d \le c \times t + s$, i.e., the distance traveled is bound from above by the number of elapsed seconds times the maximum speed plus the distance traveled during the current second.

Standard polyhedral analysis, when applied to this example, simultaneously explores both arms of the non-deterministic conditional and yields the following sequence of abstract states at node $n_1$ during the first $k$ iterations (we assume that $k < c$):

$$\{\, 0 \le s \le d \le (k-1) \times t + s,\ t + d \le k \,\}$$

The application of widening extrapolates the above sequence to $\{\, 0 \le s \le d \,\}$ (i.e., by letting $k$ go to $\infty$). Narrowing refines the result to $\{\, 0 \le s \le c,\ s \le d \,\}$. Thus, unless the widening delay is greater than $c$, the result obtained with standard analysis is imprecise.



```
volatile bool m, sec;

d = t = s = 0;

while(true)
{
   if(sec) {
     t++; s = 0;
   }
   else if(m) {
     if(s < c) {
       d++; s++;
     }
   }
}
```

(a)

(b)

$0 \le d = s \le c$
$t = 0$

(c)

$s \le d \le c \times t + s$
$0 \le d \le c$

(d)

$s \le d \le c \times t + s$
$0 \le s \le c$

(e)

**Fig. 4.** A model of a speedometer with the assumption that maximum speed is $c$ meters per second [10] ($c$ is a positive constant): (a) a program; (b) control-flow graph for the program in (a); (c) abstract state at $n_1$ after $\Pi_1^\sharp$ (edge $\langle n_1, n_2\rangle$ disabled) is analyzed; (d) abstract state at $n_1$ after $\Pi_2^\sharp$ (edge $\langle n_1, n_4\rangle$ disabled) is analyzed; (e) abstract state at $n_1$ after $\Pi_3^\sharp = \Pi^\sharp$ is analyzed.

To improve the analysis precision, we would like to analyze each of the loop's behaviors in isolation. That is, we would like to derive a sequence of program restrictions, each of which captures exactly one of the loop behaviors and suppresses the others. This can be achieved by making each program restriction enable a single outgoing edge outgoing from a node where the control is chosen non-deterministically and disable the others. After all single-behavior restrictions are processed, we can ensure that the analysis is sound by analyzing a program restriction where all of the outgoing edges are enabled.

For the program in Fig. 4(a), we construct three program restrictions: $\varPi_1^\sharp$ enables edge $\langle n_1, n_4\rangle$ and disables $\langle n_1, n_2\rangle$, $\varPi_2^\sharp$ enables edge $\langle n_1, n_2\rangle$ and disables $\langle n_1, n_4\rangle$, $\varPi_3^\sharp$ enables both edges. Figs. 4(c), 4(d), and 4(e) show the abstract states $\varTheta_1^\sharp(n_1)$, $\varTheta_2^\sharp(n_1)$, and $\varTheta_3^\sharp(n_1)$ computed by guided static analysis instantiated with the above sequence of program restrictions. Note that the overall result of the analysis in Fig. 4(e) implies the desired invariant.

We formalize the above strategy as follows. Let $V_{nd} \subseteq V$ be a set of nodes at which loop behavior is chosen. An internal state of the program transformer keeps track of which outgoing edge is to be enabled next for each node in $V_{nd}$. One particular scheme for achieving this is to make an internal state $I$ map each node $v \in V_{nd}$ to a non-negative integer: if $I(v)$ is less then the out-degree of $v$, then $I(v)$-th outgoing edge is to be enabled; otherwise, all outgoing edges are to be enabled. The initial state $I_0$ maps all nodes in $V_{nd}$ to zero.

If iteration behavior can be chosen at multiple points (e.g., the body of the loop contains a chain of non-deterministic conditionals), the following problem arises: an attempt to isolate all possible loop behaviors may generate exponentially many program restrictions. In the prototype implementation, we resort to the following heuristic: simultaneously advance the internal states for *all* reachable nodes in $V_{nd}$. This strategy ensures that the number of generated program restrictions is linear in $|V_{nd}|$; however, some loop behaviors will not be isolated.

Let $deg_{out}(v)$ denote the out-degree of node $v$; also, let $edge_{out}(v, i)$ denote the $i$-th edge outgoing from $v$, where $0 \leq i < deg_{out}(v)$. The program transformer $\varLambda_{nd}(\varPi^\sharp, I, \varTheta^\sharp)$ is defined as follows:

$$\varPi_r^\sharp(\langle u, v\rangle) = \begin{cases} \bar{\bot} & \text{if } \begin{bmatrix} u \in V_{nd},\ \varTheta^\sharp(u) \neq \bot,\ I(u) < deg_{out}(u) \\ \text{and } \langle u, v\rangle \neq edge_{out}(u, I(u)) \end{bmatrix} \\ \varPi^\sharp(\langle u, v\rangle) & \text{otherwise} \end{cases}$$

The internal state of $\varLambda_{nd}$ is updated as follows: for all $v \in V_{nd}$ such that $\varTheta^\sharp(v) \neq \bot$, $I_r(v) = I(v) + 1$; for the remaining nodes, $I_r(v) = I(v)$.

As with the first instantiation, the program transformer defined above does not satisfy the chain property. However, the sequence of program restrictions generated according to Defn. 4 is bound to stabilize in a finite number of steps. To see this, note that once node $v \in V_{nd}$ becomes reachable, at most $deg_{out}(v) + 1$ program restrictions can be generated before exhausting all of the choices for node $v$. Thus, we can enforce the chain property by making $\varLambda_{nd}$ return $\varPi^\sharp$ once the sequence of program restrictions stabilizes.

## 5 Disjunctive Extension

A single iteration of guided static analysis extends the current approximation for the entire set of reachable program states (represented with a single abstract-domain element) with the states that are reachable via the new program behaviors introduced on that iteration. However, if the abstract domain is not distributive, using a single abstract-domain element to represent the entire set of reachable program states may degrade the precision of the analysis. A more precise solution can potentially be obtained if, instead of joining together the contributions of individual iterations, the analysis represents the contribution of each iteration with a separate abstract-domain element.

In this section, we extend guided static analysis to perform such disjunctive partitioning. To isolate a contribution of a single analysis iteration, we add an extra step to the analysis. That step takes the current approximation for the set of reachable program states and constructs an approximation for the set of states that immediately exercise the new program behaviors introduced on that iteration. The resulting approximation is used as a starting point for the standard analysis run performed on that iteration. That is, an iteration of the analysis now consists of three steps: the algorithm (i) derives the (next) program restriction $\Pi_r^\sharp$; (ii) constructs an abstract-state map $\Theta_r^\sharp$ that forces a fix-point computation to explore only the new behaviors introduced in $\Pi_r^\sharp$; and (iii) performs a fix-point computation to analyze $\Pi_r^\sharp$, using $\Theta_r^\sharp$ as the initial abstract-state map.

We start by defining the *analysis history* $H_k$, a sequence of abstract-state maps obtained by the first $k \geq 0$ iterations of guided static analysis. $H_k$ maps an integer $i \in [0, k]$ to the result of the $i$-th iteration of the analysis. $H_k$ approximates the set of program states reached by the first $k$ analysis iterations: $\gamma(H_k) = \bigcup_{i=0}^{k} \gamma(H_k(i))$.

The introduction of the analysis history necessitates a change in the definition of a program transformer $\Lambda$ (Defn. 3): instead of a single abstract domain element, a program transformer must accept an analysis history as input. We leave it in the hands of the user to supply a suitable program transformer $\Lambda_{dj}$. In our implementation, we used a simple, albeit conservative way to construct such a program transformer from $\Lambda$:

$$\Lambda_{dj}(\Pi^\sharp, I, H_k) = \Lambda(\Pi^\sharp, I, \bigsqcup_{i=1}^{k} H_k(i)).$$

For the program in Fig. 1, $\Lambda_{dj}$ derives the same program restrictions as the ones derived by plain guided static analysis (see Fig. 2).

Let $\Pi_k^\sharp$ be the program restriction derived on the $k$-th iteration of the analysis, where $k \geq 1$. The set of *frontier edges* for the $k$-th iteration consists of the edges whose associated transformers are changed in $\Pi_k^\sharp$ from $\Pi_{k-1}^\sharp$ (for convenience, we define $\Pi_0^\sharp$ to map all edges to $\bar{\bot}$): $F_k = \left\{ e \in E \mid \Pi_k^\sharp(e) \neq \Pi_{k-1}^\sharp(e) \right\}$. For the program in Fig. 1, the sets of frontier edges on the second and third iterations are $F_2 = \{\langle n_1, n_3\rangle, \langle n_3, n_4\rangle\}$ and $F_3 = \{\langle n_4, n_x\rangle\}$. The *local analysis frontier* for the $k$-th iteration of the analysis is an abstract-state map that approximates the set of states that are immediately reachable via the edges in $F_k$:

$$LF_k(v) = \bigsqcup_{\langle u,v\rangle \in F_k} \left[ \bigsqcup_{i=0}^{k-1} \Pi_k^\sharp(\langle u, v\rangle)(H_{k-1}(i)(u)) \right].$$

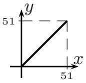
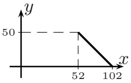
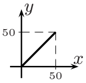
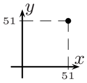
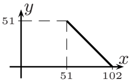
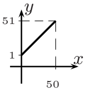
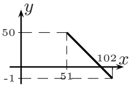
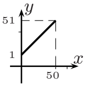
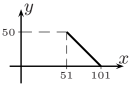
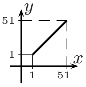
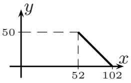
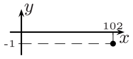
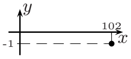
| Node | $GF_1 = \Theta_0^\sharp$ | $\Omega(\Pi_1^\sharp, GF_1)$ | $GF_2$ | $\Omega(\Pi_2^\sharp, GF_2)$ | $GF_3$ | $\Omega(\Pi_3^\sharp, GF_3)$ |
|---|---|---|---|---|---|---|
| $n_e$ | $\top$ | $\top$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ |
| $n_1$ | $\bot$ |  | $\bot$ |  | $\bot$ | $\bot$ |
| $n_2$ | $\bot$ |  | $\bot$ | $\bot$ | $\bot$ | $\bot$ |
| $n_3$ | $\bot$ | $\bot$ |  |  | $\bot$ | $\bot$ |
| $n_4$ | $\bot$ |  | $\bot$ |  | $\bot$ | $\bot$ |
| $n_5$ | $\bot$ |  | $\bot$ |  | $\bot$ | $\bot$ |
| $n_6$ | $\bot$ |  | $\bot$ |  | $\bot$ | $\bot$ |
| $n_x$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ |  |  |

**Fig. 5.** Disjunctive extension of guided static analysis: the analysis trace for the program in Fig. 1(a); for each analysis phase, the global frontier and the resulting abstract state are shown. Note that the set of abstract values computed for program point $n_x$ describes the true set of states reachable at $n_x$ (see Fig. 1(d)).

For the program in Fig. 1, the local analysis frontier on the second iteration contains a single program state: $LF_2(n_3) = \{x = y = 51\}$, which is obtained by applying the transformer associated with the edge $\langle n_1, n_3 \rangle$ to the abstract state $H_1(1)(n_1) = \{0 \leq x = y \leq 51\}$.

Some program states in the local analysis frontier may have already been explored on previous iterations. The *global analysis frontier* refines the local frontier by taking the analysis history into consideration. Ideally, we would like to compute

$$GF_k(v) = \alpha(\gamma(LF_k(v)) - \bigcup_{i=0}^{k-1} \gamma(H_{k-1}(i)(v))),$$

where "−" denotes set difference. However, this is hard to compute in practice. In our implementation, we take a simplistic approach and compute:

$$GF_k(v) = \begin{cases} \bot & \text{if } LF_k(v) \in \{H_{k-1}(i)(v) \mid 0 \leq i \leq k-1\} \\ LF_k(v) & \text{otherwise} \end{cases}$$

For the program in Fig. 1, $GF_2 = LF_2$.

**Definition 6 (Disjunctive Extension).** *Let $\Pi^\sharp$ be a program, and let $\Theta_0^\sharp$ be an abstract state that approximates the initial configuration of the program. Also, let $I_0$ be an initial internal state for the program transformer, $\Lambda_{dj}$. The disjunctive extension of guided static analysis computes the set of reachable states by performing the following iteration,*

$$H_0 = \left[0 \mapsto \Theta_0^\sharp\right] \quad \textit{and} \quad H_{i+1} = H_i \cup \left[(i+1) \mapsto \Omega(\Pi_{i+1}^\sharp, GF_{i+1})\right],$$

$$\textit{where } (\Pi_{i+1}^\sharp, I_{i+1}) = \Lambda_{dj}(\Pi^\sharp, I_i, H_i),$$

*until $\Pi_{i+1}^\sharp = \Pi^\sharp$. The result of the analysis is given by $H_{i+1}$.*

Fig. 5 illustrates the application of the disjunctive extension to the program in Fig. 1(a). The analysis precisely captures the behavior of both loop phases. Also, the abstract value computed for program point $n_x$ exactly identifies the set of program states reachable at $n_x$. Overall, the results are significantly more precise than the ones obtained with plain guided static analysis (see Fig. 3).

## 6 Experimental Results

We implemented a prototype of guided static analysis. The prototype uses a polyhedra-based numeric analysis built on top of a weighted pushdown system library, wpds++ [15], as the *base* static analysis. It relies on the Parma Polyhedral Library [2] to manipulate polyhedral abstractions. A widening delay of 4 was used in all of the experiments. The performance of each analysis run is measured in *steps*: each step corresponds to a single abstract-transformer application. Speedups (overheads) are reported as the percent of extra steps performed by the baseline analysis (evaluated analysis), respectively.

We applied the instantiation from §4.1 to the set of benchmarks that were used to evaluate policy-iteration techniques [5] and lookahead widening [11]. Tab. 1 shows the results we obtained. With the exception of "test6", the results from GSA and lookahead widening are comparable: the precision is the same, and the difference in running times can be attributed to implementation choices. This is something we expected, because GSA is a generalization of the lookahead-widening technique. However, GSA yields much better results for "test6": in "test6", the loop behavior changes when the induction variable is equal to certain values. The changes in behavior constitute short loop phases, which cause problems for lookahead widening. Also, GSA stabilizes in a fewer number of steps because simpler polyhedra arise in the course of the analysis.

**Table 1.** Experimental results: loops with multiple phases (§4.1): GSA is compared against lookahead widening (LA); Disjunctive GSA is compared against GSA. *steps* is the total number of steps performed by each of the analyses; *phases* is the number of GSA phases; *prec* reports precision improvement: "-" indicates no improvement, $k/m$ indicates that sharper invariants are obtained at $k$ out of $m$ "interesting" points (interesting points include loop heads and exit nodes)

| | LA | GSA | | | | Disjunctive GSA | | | |
|---|---|---|---|---|---|---|---|---|---|
| | steps | phases | steps | prec. | speedup(%) | phases | steps | prec. | speedup(%) |
| test1 | 58 | 2 | 54 | - | **7.9** | 2 | 42 | - | **22.2** |
| test2 | 56 | 2 | 56 | - | - | 2 | 42 | - | **25.0** |
| test3 | 58 | 1 | 44 | - | **24.1** | 1 | 42 | - | **4.5** |
| test4 | 210 | 6 | 212 | - | -1.0 | 6 | 154 | - | **27.4** |
| test5 | 372 | 3 | 368 | - | **1.1** | 3 | 406 | 1/3 | -10.3 |
| test6 | 402 | 3 | 224 | 3/3 | **44.3** | 3 | 118 | 2/3 | **47.3** |
| test7 | 236 | 3 | 224 | - | **3.4** | 3 | 154 | 4/4 | **31.3** |
| test8 | 106 | 4 | 146 | - | -37.7 | 3 | 114 | - | **21.9** |
| test9 | 430 | 4 | 444 | - | -3.3 | 4 | 488 | 4/4 | -9.9 |
| test10 | 418 | 4 | 420 | - | -0.5 | 4 | 246 | 5/5 | **41.4** |

Tab. 1 also compares the disjunctive extension to plain GSA. Because the analysis performed in each phase of the disjunctive extension does not have to reestablish the invariants obtained on previous phases, the disjunctive extension requires fewer analysis steps for most of the benchmarks. To compare the precision of the two analyses, we joined the analysis history obtained by the disjunctive extension for each program location into a single abstract value: for half of the benchmarks, the resulting abstract values are still significantly more precise than the ones obtained by plain GSA. Most notably, the two loop invariants in "test6" are further sharpened by the disjunctive extension, and the number of analysis steps is further reduced.

The instantiation in §4.2 is applied to a set of examples from [3,10]: "astree" is the (second) example that motivates the use of threshold widening in [3], "speedometer" is the example used in §4.2; the two other benchmarks are the models of a leaking

**Table 2.** Experimental results: loops with non-deterministic behavior (§4.2): *ND* $k(m)$ gives the amount of non-determinism: $k = |V_{nd}|$ and $m$ is the out-degree for nodes in $V_{nd}$; *runs* is the number of GSA runs, each run isolates iteration behaviors in different order; *steps* is the total number of analysis steps (for GSA it is the average accross all runs); *phases* is the average number of GSA phases; *inv.* indicates whether the desired invariant is obtained (for GSA, $k/m$ indicates that the invariant is obtained on $k$ out of $m$ runs)

| Program | Vars | Nodes | ND | Lookahead | | GSA | | | | Overhead |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | steps | inv. | runs | phases | steps | inv. | (%) |
| astree | 1 | 7 | 1(2) | 104 | no | 2 | 3 | 107 | **yes** | 2.9 |
| speedometer | 3 | 8 | 1(2) | 114 | no | 2 | 3 | 207 | **yes** | 81.6 |
| gas burner | 3 | 8 | 2(2) | 164 | no | 4 | 3.5 | 182.5 | 3/4 | 11.3 |
| gas burner II | 4 | 5 | 1(3) | 184 | no | 6 | 4 | 162 | 4/6 | **-12.0** |

gas burner from [10]. The results are shown in Tab. 2: guided static analysis was able to establish the desired invariants for all of the examples. We enumerated all possible orders in which iteration behaviors can be enabled for these examples. Interestingly, the precision of the analysis on the gas-burner benchmarks does depend on the order in which the behaviors are enabled. In the future, we plan to address the issue of finding optimal behavior orders.

## 7 Related Work

**Controlled state-space exploration.** Bourdoncle discusses the effect of an iteration strategy on the overall efficiency of analysis [4]. *Lazy abstraction* [14] guides the state-space exploration in a way that avoids performing joins: the CFG of a program is unfolded as a tree and stabilization is checked by a special *covering* relation. The *directed automated random testing (DART)* technique [9] restricts the analysis to the part of the program that is exercised by a particular test input; the result of the analysis is used to generate inputs that exercise program paths not yet explored. The analysis is carried out dynamically by an instrumented version of the program. Grumberg et al. construct and analyze a sequence of under-approximated models by gradually introducing process interleavings in an effort to speed up the verification of concurrent processes [12]. We believe that the GSA framework is more general than the above approaches. Furthermore, the GSA instantiations presented in this paper address the precision of widening, which is not addressed by any of the above techniques.

**Widening precision.** *Threshold widening* [3] and *widening up-to* [13] rely on external invariant guesses supplied by the user or obtained from the program code with the use of some heuristics or by running a separate analysis. In contrast, our instantiations are self-contained: that is, they do not rely on external invariant guesses. The *new control-path heuristic* [13] detects the introduction of new behaviors and delays widening until the introduced behavior is sufficiently explored. However, it lacks the ability to refine the solution for already-explored behaviors before the new behavior is introduced. Policy-iteration techniques [5,8] derive a series of program simplifications by changing the semantics of the meet operator: each simplification is analyzed with a dedicated analysis. We believe that our approach is easier to adopt because it relies on existing and well-understood analysis techniques. Furthermore, policy-iteration techniques are not yet able to operate on fully-relational abstract domains (e.g., polyhedra). The instantiation in §4.1 is the generalization of *lookahead widening* [11]: it lifts some of the restrictions imposed by lookahead widening. Gonnord et al. combine polyhedral analysis with acceleration techniques [10]: complex loop nests are simplified by "accelerating" some of the loops. The instantiation in §4.2 attempts to achieve the same effect, but does not rely on explicit acceleration techniques.

**Powerset extensions.** *Disjunctive completion* [6] improves the precision of the analysis by propagating sets of abstract-domain elements. However, to allow its use in numeric program analysis, widening operators must be lifted to operate on sets of elements [1]. Sankaranarayanan et al. [18] circumvent this problem by propagating single abstract-domain elements through an elaboration of a control-flow graph (constructed on the fly). *ESP* [7], TVLA [16], and the *trace-partitioning framework* [17] structure abstract states as functions from a specially-constructed finite set (e.g., set of FSM states [7], or set of valuations of nullary predicates [16]) into the set of abstract-domain elements: at merge points, only the elements that correspond to the same member of the set are joined. The disjunctive extension in §5 differs from these techniques in two aspects: (i) the policy for separating abstract-domain elements is imposed implicitly by the program transformer; (ii) the base-level static analysis, invoked on each iteration of GSA, always propagates single abstract-domain elements.

## References

1. Bagnara, R., Hill, P., Zaffanella, E.: Widening operators for powerset domains. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp.135–148, Springer, Heidelberg (2004)
2. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.M.: Possibly not closed convex polyhedra and the parma polyhedra library. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp.299–315, Springer, Heidelberg (2002)
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Min'e, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: The Essence of Computation: Complexity, Analysis, Transformation, pp. 85–108 (2002)
4. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Int. Conf. on Formal Methods in Prog. and their Appl, pp. 128–141 (1993)
5. Costan, A., Gaubert, S., Goubault, E., Martel, M., Putot, S.: A policy iteration algorithm for computing fixed points in static analysis of programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, Springer, Heidelberg (2005)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282 (1979)
7. Das, M., Lerner, S., Seigle, M.: Esp: Path-sensitive program verification in polynomial time. In: PLDI, pp. 57–68 (2002)
8. Gaubert, S., Goubault, E., Taly, A., Zennou, S.: Static analysis by policy interation on relational domains. In: ESOP (2007)
9. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: PLDI, pp. 213–223 (2005)
10. Gonnord, L., Halbwachs, N.: Combining widening and acceleration in linear relation analysis. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 144–160. Springer, Heidelberg (2006)
11. Gopan, D., Reps, T.: Lookahead widening. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 452–466. Springer, Heidelberg (2006)
12. Grumberg, O., Lerda, F., Strichman, O., Theobald, M.: Proof-guided underapproximation-widening for multi-process systems. In: POPL (2005)
13. Halbwachs, N., Proy, Y.-E., Roumanoff, P.: Verification of real-time systems using linear relation analysis. FMSD 11(2), 157–185 (1997)
14. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70 (2002)
15. Kidd, N., Reps, T., Melski, D., Lal, A.: WPDS++: A C++ library for weighted pushdown systems (2004), `http://www.cs.wisc.edu/wpis/wpds++/`
16. Lev-Ami, T., Sagiv, M., TVLA,: A system for implementing static analyses. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 280–301. Springer, Heidelberg (2000)
17. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: ESOP, pp. 5–20 (2005)
18. Sankaranarayanan, S., Ivancic, F., Shlyakhter, I., Gupta, A.: Static analysis in disjunctive numerical domains. In: SAS., pp. 3–17 (2006)

# Program Analysis Using Symbolic Ranges

Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta

NEC Laboratories America
{srirams,ivancic,agupta}@nec-labs.com

**Abstract.** Interval analysis seeks static lower and upper bounds on the values of program variables. These bounds are useful, especially for inferring invariants to prove buffer overflow checks. In practice, however, intervals by themselves are often inadequate as invariants due to the lack of relational information among program variables.

In this paper, we present a technique for deriving symbolic bounds on variable values. We study a restricted class of polyhedra whose constraints are stratified with respect to some variable ordering provided by the user, or chosen heuristically. We define a notion of normalization for such constraints and demonstrate polynomial time domain operations on the resulting domain of symbolic range constraints. The abstract domain is intended to complement widely used domains such as intervals and octagons for use in buffer overflow analysis. Finally, we study the impact of our analysis on commercial software using an overflow analyzer for the C language.

## 1 Introduction

Numerical domain static analysis has been used to prove safety of programs for properties such as the absence of buffer overflows, null pointer dereferences, division by zero, string usage and floating point errors [30,3,13]. Domains such as *intervals, octagons, and polyhedra* are used to symbolically over-approximate the set of possible values of integer and real-valued program variables along with their relationships under the *abstract interpretation framework* [19,8,11,21,6,25,17,27]. These domains are classified by their *precision*, i.e, their ability to represent sets of states, and *tractability*, the complexity of common operations such as union (join), post condition, widening and so on. In general, enhanced precision leads to more proofs and less false positives, while resulting in a costlier analysis.

Fortunately, applications require a domain that is "*precise enough*" rather than "*most precise*". As a result, research in static analysis has resulted in numerous trade-offs between precision and tractability. The octagon abstract domain, for instance, uses polyhedra with two variables per constraint and unit coefficients [21]. The restriction yields fast, polynomial time domain operations. Simultaneously, the pairwise comparisons captured by octagons also express and prove many common run time safety issues in practical software [3]. Nevertheless, a drawback of the octagon domain is its inability to reason with properties that may need constraints of a more complex form. Such instances arise frequently.

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 366–383, 2007.

In this paper, we study *symbolic range constraints* to discover symbolic expressions as bounds on the values of program variables. Assuming a linear ordering among the program variables, we restrict the bound for a variable $x$ to involve variables of order strictly higher than $x$. Thus, symbolic ranges can also be seen as polyhedra with triangular constraint matrices. We present important syntactic and semantic properties of these constraints including a sound but incomplete proof system derived through syntactic rewriting, and a notion of normalization under which the proof system is complete. Using some basic insights into the geometry of symbolic range constraints, we study algorithms for the various domain operations necessary to carry out program verification using symbolic range constraints. We also study the practical impact of our domain on large programs including performance comparisons with other domains.

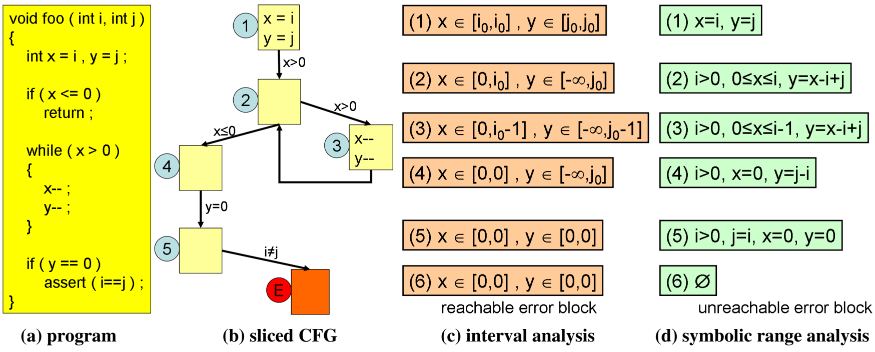
*Related work.* Range analysis has many applications in program verification and optimization. Much work has focused on the interval domain and its applications. Cousot & Cousot present an abstract interpretation scheme for interval analysis using widening and narrowing [8]. Recent work has focused on the elimination of widenings/narrowings in the analysis using linear programming [23], rigorous analysis of the data flow equations [28], and *policy iteration* [7,14].

Blume & Eigenmann study symbolic ranges for applications in compiler optimizations [4]. Their approach allows ranges that are *non-linear* with multiplication and max/min operators. However, the presence of non-linearity leads to domain operations of exponential complexity. Whereas polynomial time operations are derived heuristically, the impact of these heuristics on precision is unclear. Even though some aspects of our approach parallel that of Blume *et al.*, there are numerous fundamental differences: we focus on range constraints that are always linear, convex and triangulated based on a single, explicit variable ordering. These restrictions vastly simplify the design and implementation of domain operations while providing some insights into the properties of symbolic range constraints. Finally, we provide an experimental evaluation of the efficacy of our domain for eliminating array bounds checks in practical examples.

Symbolic ranges may also be obtained using the LP-based approach of Rugina & Rinard [23,32]. The bounds obtained relate the current value of a variable and the values of parameters at the function entry. The advantage of this approach is its freedom from heuristics such as widening/narrowing. However, the LP-formulation is based on a weaker proof system for generating consequences of linear inequalities by directly comparing coefficients, and potentially generates weaker invariants.

Symbolic range constraints are also used in the path-based analysis tool ARCHER due to Xie et al. [31]. However, the bounding expressions used in their approach can involve at most one other variable.

We illustrate symbolic ranges for invariant computation using a motivating example presented in Fig. 1(a). Assuming that the analysis starts at the function `foo`, we analyze whether the assertion at the end of the function holds. Fig. 1(b) shows the control flow graph for this example after program slicing. Fig. 1(c) shows an interval analysis computation for this example. In this example, the



```
void foo ( int i, int j )
{
   int x = i , y = j ;

   if ( x <= 0 )
      return ;

   while ( x > 0 )
   {
      x-- ;
      y-- ;
   }

   if ( y == 0 )
      assert ( i==j ) ;
}
```

1
x = i
y = j
x>0
2
x>0
3
x--
y--
x≤0
4
y=0
5
i≠j
E
(1) $x \in [i_0,i_0]$ , $y \in [j_0,j_0]$
(2) $x \in [0,i_0]$ , $y \in [-\infty,j_0]$
(3) $x \in [0,i_0-1]$ , $y \in [-\infty,j_0-1]$
(4) $x \in [0,0]$ , $y \in [-\infty,j_0]$
(5) $x \in [0,0]$ , $y \in [0,0]$
(6) $x \in [0,0]$ , $y \in [0,0]$
reachable error block
(1) x=i, y=j
(2) i>0, 0≤x≤i, y=x-i+j
(3) i>0, 0≤x≤i-1, y=x-i+j
(4) i>0, x=0, y=j-i
(5) i>0, j=i, x=0, y=0
(6) ∅
unreachable error block

**(a) program** **(b) sliced CFG** **(c) interval analysis** **(d) symbolic range analysis**

**Fig. 1.** A motivating example

interval analysis is not powerful enough to conclude that the assertion can never be violated.

Consider the analysis using symbolic ranges, for the variable ordering `i`,`j`, `x`, `y` (see Fig. 1(d)). Since symbolic ranges can represent the loop invariant `y`=`x-i+j`, the analysis discovers that for `x`=`y`=0 this implies `i`=`j` at the point of the assertion. Note also that this assertion cannot be proved using octagons, since the loop invariant is not expressible in terms of octagonal relationships.

## 2 Preliminaries

We assume that all program variables are conservatively modeled as reals. Our analysis model does not consider features such as complex data structures, procedures and modules. These may be handled using well-known extensions [22]. Let $C$ be the first order language of assertions over free variables $\boldsymbol{x}$, and $\models \subseteq C \times C$ denote entailment. An assertion $\varphi$ represents a set of models $[[\varphi]]$.

**Definition 1 (Control Flow Graph).** *A Control Flow Graph (CFG)* $\Pi : \langle \boldsymbol{x}, L, E, \mathsf{c}, \mathsf{u}, \ell_0 \rangle$ *consists of variables* $\boldsymbol{x} = \langle x_1, \ldots, x_n \rangle$*, locations* $L$ *and edges* $E$ *between locations. Each edge* $E$ *is labeled by a condition* $\mathsf{c}(e) \in C$*, and an update* $\mathsf{u}(e) : \ \boldsymbol{x} \ := \ f(\boldsymbol{x})$*.* $\ell_0 \in L$ *is the* start location.

A state of the program is a tuple $\langle \ell, \boldsymbol{a} \rangle$ where $\ell \in L$ is a location and $\boldsymbol{a}$ represents a valuation of the program variables $\boldsymbol{x}$. Given a CFG $\Pi$, an *assertion map* $\eta : L \mapsto C$ is a function mapping each location $\ell \in L$ to an assertion $\eta(\ell) \in C$. An assertion map characterizes a set of states $\langle \ell, \boldsymbol{a} \rangle$ such that $\boldsymbol{a} \in [[\eta(\ell)]]$. Let $\eta_1 \models \eta_2$ iff $\forall \ell \in L, \ \eta_1(\ell) \models \eta_2(\ell)$. Given an assertion $\varphi$ and an edge $e : \ \ell \rightarrow m \in E$, the (concrete) post-condition of $\varphi$ wrt $e$, denoted $post(\varphi, e)$ is given by the first order assertion $post(\varphi, e) : \ (\exists \ \boldsymbol{x}_0) \ \varphi[\boldsymbol{x}_0] \ \wedge \ \mathsf{c}(e)[\boldsymbol{x}_0] \wedge \boldsymbol{x} = \mathsf{u}(e)[\boldsymbol{x}_0]$.

**Definition 2 (Inductive Assertion Map).** *An assertion map* $\eta$ *is* inductive *iff (a)* $\eta(\ell_0) \equiv true$*, and (b) for all edges* $e : \ell \rightarrow m$*,* $post(\eta(\ell), e) \models \eta(m)$*.*

A safety property $\Gamma$ is an assertion map labeling each location with a property to be verified. In order to prove a safety property $\Gamma$, we find an inductive assertion map $\eta$, such that $\eta \models \Gamma$. "*Concrete interpretation*" can be used to construct the inductive invariant map. Consider an *iterative sequence* of assertion maps $\eta^0, \eta^1, \ldots, \eta^N, \cdots$.

$$\eta^0(\ell) = \begin{cases} \mathit{true}, & \ell = \ell_0, \\ \mathit{false}, & \text{otherwise.} \end{cases} \quad \text{and } \eta^{i+1}(\ell) = \eta^i(\ell) \ \vee \bigvee_{e:\ m \to \ell} \mathit{post}(\eta^i(m), e)$$

Note that $\eta^i \models \eta^{i+1}$. The iteration *converges* if $(\exists N > 0)\ \eta^{N+1} \models \eta^N$. If the iteration converges in $N > 0$ (finitely many) steps, the result $\eta^N$ is an inductive assertion. However, the iteration may not converge for all programs. Furthermore, detecting convergence is undecidable, in general. As a result, concrete interpretation, as shown above, is impractical for programs. Therefore, we over-approximate the *concrete interpretation* in a suitable *abstract domain* [9,10].

*Abstract domains.* An abstract domain is a bounded lattice $\langle A, \sqsubseteq, \sqcap, \sqcup, \top, \bot \rangle$. It is useful to think of $A$ as an assertion language and $\sqsubseteq$ as an entailment relation. The meet $\sqcap$ and the join $\sqcup$ are approximations of the logical conjunction and disjunction respectively. Formally, we require functions $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ known as the *abstraction* and *concretization* functions resp. that form a *Galois connection* (see [9,10] for a complete description). An abstract post condition operator $\mathit{post}_A(a, e)$ over-approximates the concrete post condition such that for all $a \in A$, $\mathit{post}(\gamma(a), e) \models \gamma(\mathit{post}_A(a, e))$. An *abstract domain map* $\pi : L \mapsto A$ maps each location $\ell \in L$ to an abstract element $\pi(\ell)$. The concrete iteration sequence is generalized to yield an abstract iteration sequence:

$$\pi^0(\ell) = \begin{cases} \top, & \text{if } \ell = \ell_0 \\ \bot, & \text{otherwise} \end{cases} \quad \mathit{and}\ \ \pi^{i+1}(\ell) = \pi^i(\ell) \ \sqcup \bigsqcup_{e:\ m \to \ell} \mathit{post}_A(\pi^i(m), e)\,.$$

Again, $\pi^i \sqsubseteq \pi^{i+1}$, and the iteration converges if $\exists N > 0$ s.t. $\pi^{N+1} \sqsubseteq \pi^N$. If convergence occurs then it follows that $\gamma \circ \pi^N$ is an inductive assertion. If the lattice $A$ is of finite height or satisfies the *ascending chain condition*, convergence is always guaranteed. On the other hand, many of the domains commonly used in program verification do not exhibit these conditions. Convergence, therefore, needs to be forced by the use of *widening*.

Formally, given $a_1, a_2$, their widening $a_1 \nabla a_2$ satisfies $a_2 \sqsubseteq (a_1 \sqcup a_2)$. Additionally, given an infinite sequence of objects $a_1, \ldots, a_m, \ldots$, the *widened sequence* given by $b_0 = \bot, \text{ and } b_{i+1} = b_i \nabla (b_i \sqcup a_i)$, converges in finitely many steps. In summary, the abstract iteration requires the following operations: (a) *Join* $\sqcup$ (*meet* $\sqcap$) over-approximates the logical or (and), (b) *Abstract post condition* $\mathit{post}_A$ over-approximates *post*, (c) *Inclusion test* $\sqsubseteq$ to check for the termination of the iteration, and (d) *Widening* operator $\nabla$ to force convergence. In practice, we also require other operations such as projection and narrowing.

## 3 Symbolic Range Constraints

Let $\mathcal{R}$ represent the reals and $\mathcal{R}^+$, the set of *extended reals* ( $\mathcal{R} \cup \{\pm\infty\}$). Let $\boldsymbol{x}$ denote a vector of $n > 0$ real-valued variables. The $i^{\text{th}}$ component of the vector $\boldsymbol{x}$ is written $x_i$. We use $A, B, C$ to denote matrices. Throughout this section, we fix a variable ordering given by $x_1 \prec x_2 \prec \cdots \prec x_n$, with the index $i$ of a variable $x_i$ being synonymous with its rank in this ordering.

A *linear expression* is of the form $\mathsf{e} : \ \boldsymbol{c}^T\boldsymbol{x} + d$ where $\boldsymbol{c}$ is a vector of coefficients over the reals, while $d \in \mathcal{R}^+$ is the constant coefficient. By convention, a linear expression of the form $c^T\boldsymbol{x} \pm \infty$ is identical to $\mathbf{0}^T\boldsymbol{x} \pm \infty$. For instance, the expression $2x_1 + \infty$ is identical to $0x_1 + \infty$. A *linear inequality* is of the form $\mathsf{e} \bowtie 0$, where $\bowtie \in \{\geq, \leq, =\}$. A *linear constraint* is a conjunction of finitely many linear inequalities $\varphi : \ \bigwedge_i \mathsf{e}_i \geq 0$.

Given an inequality $e \geq 0$, where $e$ is not a constant, its *lead variable* $x_i$ is the least index $i$ s.t. $c_i \neq 0$. We may write such an inequality in the *bounded form* $x_i \ \widehat{\bowtie} \ \mathsf{e}_i$, where $x_i$ is the lead variable and $\mathsf{e}_i = \frac{1}{c_i}\mathsf{e} - x_i$. The sign $\widehat{\bowtie}$ denotes the reversal of the direction of the inequality if $c_i < 0$. As an example, consider the inequality $2x_2 + 3x_5 + 1 \leq 0$. Its lead variable is $x_2$ and bounded form is $x_2 \leq -\frac{3}{2}x_5 - \frac{1}{2}$. We reuse the $\models$ relation to denote entailment among linear constraints in the first order theory of linear arithmetic.

**Definition 3 (Symbolic Range Constraint).** *A symbolic range constraint (*SRC*) is of the form $\varphi : \ \bigwedge_{i=1}^{n} \mathsf{l}_i \leq x_i \leq \mathsf{u}_i$ where for each $i \in [1, n]$, the linear expressions $\mathsf{l}_i, \mathsf{u}_i$ are made up of variables in the set $\{x_{i+1}, \ldots, x_n\}$. In particular, $\mathsf{l}_n, \mathsf{u}_n$ are constants. The linear assertions false and true are also assumed to be* SRC*s.*

The absence of a bound for $x_j$ is modeled by setting the bound to $\pm\infty$. Given an SRC $\varphi : \ \bigwedge_{j=1}^{n} \mathsf{l}_j \leq x_j \leq \mathsf{u}_j$, let $\varphi_{[\mathsf{i}]}$ denote the assertion $\bigwedge_{j=i}^{n} \mathsf{l}_j \leq x_j \leq \mathsf{u}_j$.

*Example 1.* $\varphi : \ x_2{+}4 \leq x_1 \leq 2x_3{+}x_2{+}4 \wedge -x_3 \leq x_2 \leq x_3{+}4 \wedge -\infty \leq x_3 \leq 0$ is a SRC. The variable ordering is $x_1 \prec x_2 \prec x_3$. The bound for $x_1$ involves $\{x_2, x_3\}$, $x_2$ involves $\{x_3\}$ and $x_3$ has constant bounds.

*Implied constraints & normalization.* Given a symbolic range $\mathsf{l}_i \leq x_i \leq \mathsf{u}_i$, its *implied inequality* is $\mathsf{l}_i \leq \mathsf{u}_i$. Note that the implied inequality $\mathsf{l}_i \leq \mathsf{u}_i$ only involves variables $x_{i+1}, \ldots, x_n$.

**Definition 4 (Normalization).** *A* SRC *is* normalized *iff for each variable bound $\mathsf{l}_i \leq x_i \leq \mathsf{u}_i$, $\varphi_{[\mathsf{i}+1]} \models \mathsf{l}_i \leq \mathsf{u}_i$. By convention, the empty and universal* SRC *are normalized.*

*Example 2.* The SRC $\varphi$ from Example 1 is not normalized. The implied constraint $0 \leq 2x_3$ derived from the range $x_2 + 4 \leq x_1 \leq 2x_3 + x_2 + 4$ is not implied by $\varphi_{[2]}$. The equivalent SRC $\varphi'$ is normalized:

$$\varphi' : \ x_2 + 4 \leq x_1 \leq 2x_3 + x_2 + 4 \ \wedge \ -x_3 \leq x_2 \leq x_3 + 4 \ \wedge \ 0 \leq x_3 \leq 0$$

Unfortunately, not every SRC has a normal equivalent. The SRC $\psi: \ x_2 - x_3 \leq x_1 \leq 1 \ \wedge \ 0 \leq x_2 \leq 2 \ \wedge \ 0 \leq x_3 \leq 2$ forms a counter-example. The projection of $\psi$ on the $\{x_2, x_3\}$ is a five sided polygon, whereas any SRC in 2D is a *trapezium.*

*Weak optimization algorithms.* Optimization is used repeatedly as a primitive for other domain operations including abstraction, join and intersection. Consider the optimization instance min. $(\mathsf{e} : \boldsymbol{c}^T\boldsymbol{x} + d)$ s.t. $\varphi$. Let $\varphi$ be a satisfiable SRC with bound $\mathsf{l}_j \leq x_j \leq \mathsf{u}_j$ for index $0 \leq j < n$. We let $e \xrightarrow{\varphi,j} e'$ denote the replacement of $x_j$ in $e$ by $\mathsf{l}_j$ (lower bound in $\varphi$) if its coefficient in $\mathsf{e}$ is positive, or $\mathsf{u}_j$ otherwise.

$$\text{Formally, } \mathsf{e}' = \begin{cases} \mathsf{e} - c_j x_j + c_j \mathsf{l}_j, \ c_j \geq 0, \\ \mathsf{e} - c_j x_j + c_j \mathsf{u}_j, \ c_j < 0. \end{cases}$$

The *canonical sequence*, given by $\mathsf{e} \xrightarrow{\varphi,1} \mathsf{e}_1 \cdots \xrightarrow{\varphi,n} \mathsf{e}_n$, replaces variables in the ascending order of their indices. The canonical sequence, denoted in short by $\mathsf{e} \overset{\varphi}{\twoheadrightarrow} \mathsf{e}_n$, is unique, and yields a unique result. The following lemma follows from the triangularization of SRCs:

**Lemma 1.** *For the canonical sequence* $\mathsf{e} \xrightarrow{\varphi,1} \cdots \xrightarrow{\varphi,n} \mathsf{e}_n$*, each intermediate expression* $\mathsf{e}_i$ *involves only the variables in* $\{x_{i+1}, \ldots, x_n\}$*. Specifically,* $\mathsf{e}_n \in \mathcal{R}^+$*.*

*Example 3.* Consider the SRC $\varphi'$ defined in Example 2 and the expression $\mathsf{e}$ : $-3x_1 + 2x_2 + 8x_3$. This yields the sequence $-3x_1 + 2x_2 + 8x_3 \xrightarrow{\varphi',1} -x_2 + 2x_3 - 12 \xrightarrow{\varphi',2} x_3 - 16 \xrightarrow{\varphi',3} -16$.

It follows that $\mathsf{e}_n$ under-approximates the minima of the optimization problem, and if $\varphi$ is normalized, weak optimization computes the exact minima; the same result as any other LP solver.

**Theorem 1 (Weak Optimization Theorem).** *Given a constraint* $\varphi$ *and the sequence* $e \overset{\varphi}{\twoheadrightarrow} \mathsf{e}_n$, $\varphi \models \mathsf{e} \geq \mathsf{e}_n$*. Furthermore, if* $\varphi$ *is normalized then* $\mathsf{e}_n =$ min $\mathsf{e}$ s.t. $\varphi$*.*

Weak optimization requires $O(n)$ rewriting steps, each in turn involving arithmetic over expressions of size $O(n)$. Therefore, the complexity of weak optimization for a SRC with $n$ constraints is $O(n^2)$.

*Example 4.* From Theorem 1, $-16$ is the exact minimum in Example 3. Consider the equivalent constraint $\varphi$ from Example 1. The same objective minimizes to $-\infty$ (unbounded) if performed w.r.t. $\varphi$.

Optimization provides an inference mechanism: given $d = \min \ \mathsf{e}$ s.t. $\varphi$, we infer $\varphi \models \mathsf{e} \geq d$. By Theorem 1, an inference using weak optimization is always sound. It is also complete, if the constraint $\varphi$ is also normalized. Given SRC $\varphi$, we write $\varphi \models_W \mathsf{e} \geq 0$ to denote inference of $\mathsf{e} \geq 0$ from $\varphi$ by weak optimization. Similarly, $\varphi \models_W \bigwedge_i \mathsf{e}_i \geq 0$ iff $(\forall \ i) \ \varphi \models_W \mathsf{e}_i \geq 0$.

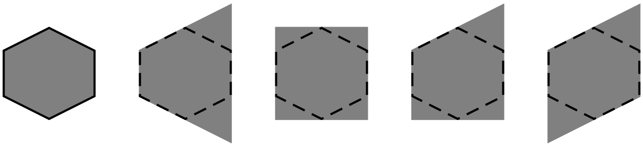


**Fig. 2.** Four possible SRC abstractions of a 2D hexagon (among many others)

Optimization for SRCs can also be solved by efficient algorithms such as SIMPLEX or interior point techniques. We will henceforth refer to such techniques as *strong optimization* techniques. In practice, however, we prefer weak optimization since (a) it out-performs LP solvers, (b) is less dependent on floating point arithmetic, and (c) allows us to draw sound inferences wherever required. As a curiosity, we also note that well-known examples such as Klee-Minty cubes and Goldfarb cubes that exhibit worst case behavior for SIMPLEX algorithms happen to be SRCs [5]. It is unclear if such SRCs will arise in practical verification problems. For the rest of the paper, we will assume optimization is always performed using weak optimization. Nevertheless, any call to weak optimization can be substituted by a call to strong optimization. Experimental results shown in Section 6 provide further justification for this choice.

We also use optimization to compare expressions wrt a given SRC $\varphi$. We write $\mathsf{e}_1 \gg_\varphi \mathsf{e}_2$ iff $\varphi \models_W \mathsf{e}_1 \geq \mathsf{e}_2$. Expressions are equivalent, written $\mathsf{e}_1 \equiv_\varphi \mathsf{e}_2$, if $\varphi \models \mathsf{e}_1 = \mathsf{e}_2$, and incomparable, denoted $\mathsf{e}_1 \Diamond_\varphi \mathsf{e}_2$, if neither inequality holds.

*Abstraction.* The abstraction function converts arbitrary first-order formulae to symbolic ranges. In practice, programs we analyze are first *linearized.* Therefore, abstraction needs to be defined only on polyhedra. Abstraction is used as a primitive operation that organizes arbitrary linear constraints into the form of SRCs.

Let $\psi$ be a polyhedron represented as a conjunction of linear inequalities $\bigwedge_i \mathsf{e}_i \geq 0$. We seek a SRC $\varphi : \alpha(\psi)$ such that $\psi \models \varphi$. Unfortunately, this SRC abstraction $\alpha(\psi)$ may not be uniquely defined. Figure 2 shows possible SRC abstractions of a hexagon in 2 dimensions that are all semantically incomparable.

Abstraction of a given polyhedron $\psi$ is performed by sequentially inserting the inequalities of $\psi$ into a target SRC, starting initially with the SRC *true*. The result is an SRC $\alpha(\psi)$.

*Inequality Insertion.* Let $\varphi$ be a SRC and $\mathsf{e}_j \geq 0$ be an inequality. As a primitive we consider the problem of deriving an abstraction $\alpha(\varphi \wedge \mathsf{e}_j \geq 0)$. We consider the case wherein $x_j \leq \mathsf{b}_j$ is the bounded form of $\mathsf{e}_j$. The case where the bounded form is $x_j \geq \mathsf{b}_j$ is handled symmetrically. Also, let $\mathsf{l}_j \leq x_j \leq \mathsf{u}_j$ be the existing bounds for $x_j$ in $\varphi$.

Using expression comparison, we distinguish three cases, (a) $\mathsf{b}_j \gg_{\varphi_{[j+1]}} \mathsf{u}_j$ , (b) $\mathsf{u}_j \gg_{\varphi_{[j+1]}} \mathsf{b}_j$ and (c) $\mathsf{u}_j \Diamond_{\varphi_{[j+1]}} \mathsf{b}_j$, as depicted in Figure 3. For case (a), the bound $x_j \leq \mathsf{u}_j$ entails $x_j \leq \mathsf{b}_j$, therefore we need not replace $\mathsf{u}_j$. The reverse holds for case (b), and $\mathsf{u}_j$ is replaced. However, for case (c), neither bound entails the other. We call this a *conflict.*

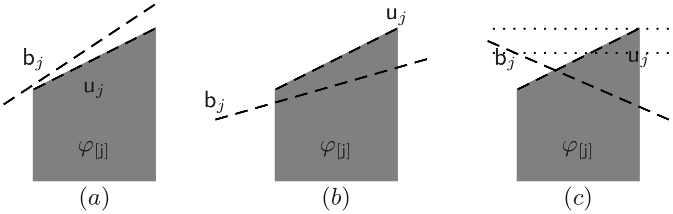


**Fig. 3.** Three cases encountered during abstraction. (a) $\mathsf{b}_j \gg_\varphi \mathsf{u}_j$, (b) $\mathsf{u}_j \gg_\varphi \mathsf{b}_j$ and (c) $\mathsf{u}_j \Diamond \mathsf{b}_j$ showing a conflict.

A *conflict* forces us to choose between two bounds $\mathsf{u}_j, \mathsf{b}_j$ where neither is semantically stronger than the other. Conflicts are due to the lack of a unique SRC abstraction. We handle conflicts using *conflict resolution heuristics* provided by the user. We describe a few possible heuristics below.

**Interval Heuristic.** We consider the worst case interval bound on $x_j$ resulting from either choice of bounds. Let $c = \max\ \mathsf{b}_j\ \mathsf{s.t.}\ \ \varphi_{[j+1]}$ and similarly, $d = \max\ \mathsf{u}_j\ \mathsf{s.t.}\ \ \varphi_{[j+1]}$. If $c < d$, we replace $\mathsf{u}_j$ by $\mathsf{b}_j$, and retain $\mathsf{u}_j$ otherwise. Figure 3(c) shows a geometric interpretation.

**Metric.** Choose the bound that minimizes the volume of the resulting SRC, or alternatively, the distance from a reference set.

**LexOrder.** Choose syntactically according to lexicographic order.

**Fixed.** Always choose to retain the original bound $\mathsf{u}_j$, or replace it with $\mathsf{b}_j$.

The result of abstraction is not guaranteed to be normalized. If there are no conflicts in the abstraction process then semantic equivalence of the SRC to the original polyhedron follows. In summary, the abstraction algorithm is parameterized by the conflict resolution heuristic. Our implementation uses the interval heuristic to resolve conflicts and the lexicographic order to break ties. Let $\alpha$ denote the abstraction function that uses some conflict resolution strategy.

**Lemma 2.** *For a constraint $\psi$, $\alpha(\psi)$ is a* SRC *and $\psi \models \alpha(\psi)$.*

Each inequality insertion requires us to solve finitely many optimization problems. Weak optimization requires time $O(n^2)$. Therefore, the SRC abstraction a polyhedron with $m$ inequalities can be computed in time $O(n^2 m)$.

## 4 Domain Operations

The implementation of various operations required for static analysis over SRCs is discussed in this section.

*Forced normalization.* A SRC $\varphi$ may fail to be normalized in the course of our analysis as a result of abstraction or other domain operations. Failure of normalization can itself be detected in $O(n^3)$ time using weak optimization using the lemma below:

**Lemma 3.** *A* SRC $\varphi$ *is normalized iff for each bound* $\mathsf{l}_i \leq x_i \leq \mathsf{u}_i$, $0 \leq i < n$, $\varphi_{[i+1]} \models_W \mathsf{l}_i \leq \mathsf{u}_i$. *Note that the* $\models_W$ *relation is sufficient to test normalization.*

Bottom-up: In general, a SRC that is not normalized may not have a normal equivalent. However, it is frequently the case that normalization may be achieved by simply propagating missing information from lower order indices up to the higher order indices. We consider each bound $\mathsf{l}_j \leq x_j \leq \mathsf{u}_j$, for $j = n-1, \ldots, 1$, and insert the implied inequality $\mathsf{l}_j \leq \mathsf{u}_j$ into $\varphi_{[j+1]}$ using the abstraction procedure described in Section 3. This process does not always produce a normalized constraint. However, the procedure itself is useful since it can sometimes replace missing bounds for variables by using a bound implied by the remaining constraints.

*Example 5.* Recall the SRC $\varphi: \ x_2 + 4 \leq x_1 \leq 2x_3 + x_2 + 4 \ \wedge \ -x_3 \leq x_2 \leq x_3 + 4 \ \wedge \ -\infty \leq x_3 \leq 0$ from Example 1. The implied inequality $x_2 + 4 (\leq x_1) \leq 2x_3 + x_2 + 4$ simplifies to $x_3 \geq 0$. When inserted, this yields the normalized SRC $\varphi'$ from Example 2.

Even though bottom-up normalization is not always guaranteed to succeed, it generally improves the result of the weak optimization algorithm. We therefore employ it after other domain operations as a pre-normalization step.

Top-down: Add constant offsets $\alpha_j, \beta_j > 0$ to bounds $\mathsf{l}_j, \mathsf{u}_j$ such that the resulting bounds $\mathsf{l}_j - \alpha_j \leq x_j \leq \mathsf{u}_j + \beta_j$ are normalized. In practice, $\alpha_j, \beta_j$ may be computed by recursively normalizing $\varphi_{[j+1]}$ and then using weak optimization. As a corollary of Lemma 3, top-down technique always normalizes.

**Lemma 4.** *Let* $\varphi$ *be an* SRC *and* $\varphi_1, \varphi_2$ *be the results of applying bottom-up and top-down techniques, respectively to* $\varphi$. *It follows that* $\varphi \models \varphi_1$ *and* $\varphi \models_W \varphi_2$. *However,* $\varphi \models_W \varphi_1$ *does not always hold.*

Following other numerical domains, we note that normalization should never be forced after a widening operation to ensure termination [21].

*Intersection & join.* Given two SRCs $\varphi_1 \wedge \varphi_2$ their intersection can be performed by using the abstraction procedure, i.e., $\varphi_1 \sqcap \varphi_2 = \alpha(\varphi_1 \wedge \varphi_2)$. In general, the best possible join $\varphi_1 \sqcup \varphi_2$ for SRCs $\varphi_1, \varphi_2$ can be defined as the abstraction of the polyhedral convex hull $\varphi_1, \varphi_2$. However, convex hull computations are expensive, even for SRCs. We describe a direct generalization of the interval join used for value ranges. Let $\mathsf{l}_j \leq x_j \leq \mathsf{u}_j$ be a bound in $\varphi_1$ (similar analysis is used for bounds in $\varphi_2$). Consider the following optimization problems: $c_j^1 = \mathsf{min.} \ \ x_j - \mathsf{l}_j \ \mathsf{s.t.} \ \ \varphi_2$, $d_j^1 = \mathsf{max.} \ \ x_j - \mathsf{u}_j \ \mathsf{s.t.} \ \ \varphi_2$.

Note that $\varphi_2 \models \mathsf{l}_j + c_j^1 \leq x_j \leq \mathsf{u}_j + d_j^1$, while $\varphi_1 \models \mathsf{l}_j + 0 \leq x_j \leq \mathsf{u}_j + 0$. As a result, $(\varphi_1 \sqcup \varphi_2) \models \mathsf{l}_j + \min(c_j^1, 0) \leq x_j \leq \mathsf{u}_j + \max(0, d_j^1)$. We call such a constraint the *relaxation* of $x_j$ in $\varphi_1$. Let $\varphi_{12}$ be the result of relaxing each bound in $\varphi_1$ wrt $\varphi_2$. Similarly, let $\varphi_{21}$ be obtained by relaxing each bound in $\varphi_2$ wrt $\varphi_1$. We define the range join as $\varphi_1 \sqcup_{\mathrm{R}} \varphi_2 : \ \varphi_{12} \sqcap \varphi_{21}$.

**Lemma 5.** *Given any* SRC $\varphi_1, \varphi_2$, $\varphi_i \models_W \varphi_1 \sqcup_{\mathrm{R}} \varphi_2$, $i = 1, 2$. *Also,* $\varphi_1 \sqcap \varphi_2 \models \varphi_i$. *However, this containment may not be provable using* $\models_W$.

Relaxing each constraint requires $O(n)$ optimization, each requiring $O(n^2)$ time. Finally, abstraction itself requires $O(n^3)$ time. As a result join can be achieved in time $O(n^3)$.

*Example 6.* Consider the SRCs $\varphi_1, \varphi_2$ shown below:

$$\varphi_1 : \left\{ \begin{array}{c} x_2 \leq x_1 \leq 2x_2 + 4 \\ x_3 \leq x_2 \leq 5 \\ -4 \leq x_3 \leq 4 \end{array} \right\} \quad \varphi_2 : \left\{ \begin{array}{c} -\infty \leq x_1 \leq x_2 \\ 0 \leq x_2 \leq x_3 + 1 \\ 0 \leq x_3 \leq 2 \end{array} \right\}$$

The relaxed constraints are given by

$$\varphi_{12} : \left\{ \begin{array}{c} -\infty \leq x_1 \leq 2x_2 + 4 \\ x_3 - 2 \leq x_2 \leq 5 \\ -4 \leq x_3 \leq 4 \end{array} \right\} \quad \varphi_{21} : \left\{ \begin{array}{c} -\infty \leq x_1 \leq x_2 + 9 \\ -4 \leq x_2 \leq x_3 + 9 \\ -4 \leq x_3 \leq 4 \end{array} \right\}$$

The join is computed by intersecting these constraints:

$$\varphi : \ -\infty \leq x_1 \leq 2x_2 + 4 \ \wedge \ x_3 - 2 \leq x_2 \leq 5 \ \wedge \ -4 \leq x_3 \leq 4 \,.$$

*Projection.* Projection is an important primitive for implementing the transfer function across assignments and modeling scope in interprocedural analysis. The "best" projection is, in general, the abstraction of the projection carried out over polyhedra. However, like convex hull, polyhedral projection is an exponential time operation in the worst case.

**Definition 5 (Polarity).** *A variable $z$ occurring in the RHS of a bound $x_j \bowtie \mathsf{b}_j$ has* positive polarity *if $\mathsf{b}_j$ is a lower bound and $z$ has a positive coefficient, or $\mathsf{b}_j$ is an upper bound and $z$ has a negative coefficient. The variable has negative polarity otherwise. Variable $z$ with positive polarity in a constraint is written $z^+$, and negative polarity as $z^-$ (see Example 7 below).*

*Direct projection.* Consider the projection of $x_j$ from SRC $\varphi$. Let $\mathsf{l}_j \leq x_j \leq \mathsf{u}_j$ denote the bounds for the variable $x_j$ in $\varphi$. For an occurrence of $x_j$ in a bound inequality of the form $x_i \bowtie \mathsf{b}_i : \ \boldsymbol{c}^T \boldsymbol{x} + d$ (note $i < j$ by triangulation), we replace $x_j$ in this expression by one of $\mathsf{l}_j, \mathsf{u}_j$ based on the *polarity replacement rule*: occurrences of $x_j^+$ are replaced by the lower bound $\mathsf{l}_j$, and $x_j^-$ are by $\mathsf{u}_j$. Finally, $x_j$ and its bounds are removed from the constraint. Direct projection can be computed in time $O(n^2)$.

**Lemma 6.** *Let $\varphi'$ be the result of a simple projection of $x_j$ from $\varphi$. It follows that $\varphi'$ is an* SRC *and $(\exists x_j) \ \varphi \models \varphi'$.*

*Example 7.* Direct projection of $z$ from $\varphi : \ z^+ \leq x \leq z^- + 1 \ \wedge \ z^+ - 2 \leq y \leq z^- + 3 \ \wedge \ -\infty \leq z \leq 5$, replaces $z^+$ with $-\infty$ and $z^-$ with 5 at each occurrence, yielding $\varphi' : \ -\infty \leq x \leq 6 \ \wedge \ -\infty \leq y \leq 8$.

*Indirect projection.* Direct projection can be improved by using a simple modification of *Fourier-Motzkin* elimination technique.

A *matching pair* for the variable $x_j$ consists of two occurrences of variable $x_j$ with opposite polarities in bounds $x_i \bowtie \alpha_j x_j^+ + \mathsf{e}_i$ and $x_k \bowtie \alpha_j x_j^- + \mathsf{e}_k$ with $i \neq k$. The matching pairs for the SRC $\varphi$ from Example 7 are:

$$\varphi : \left\{ \boxed{z^+} \leq x \leq \boxed{z^-} + 1 \ \wedge \ \boxed{z^+} - 2 \leq y \leq \boxed{z^-} + 3 \ \wedge \ -\infty \leq z \leq 5 \right\}$$

There are two matching pairs for the variable $z$ shown using arrows. The matching pair $z^+ \leq x$ and $y \leq z^- + 3$ can be used to rewrite the former constraint as $y - 3 \leq x$. Similarly the other matching pair can be used to rewrite the upper bound of $x$ to $x \leq y + 2$. An indirect projection of the constraint in Example 7, using matching pairs yields the result $y - 3 \leq x \leq y + 3 \ \wedge \ -\infty \leq y \leq 8$.

Matching pairs can be used to improve over direct projection, especially when the existing bounds for the variables to be projected may lead to too coarse an over-approximation. They are sound and preserve the triangular structure.

*Substitution.* The substitution $x_j \mapsto \mathsf{e}$ involves the replacement of every occurrence of $x_j$ in the constraint by $\mathsf{e}$. In general, the result of carrying out the replacements is not a SRC. However, the abstraction algorithm can be used to reconstruct a SRC as $\varphi' : \ \alpha(\varphi[x \mapsto \mathsf{e}])$.

*Transfer function.* Consider a SRC $\varphi$ and an assignment $x_j \ := \ \mathsf{e}$, where $e \equiv \boldsymbol{c}^T \boldsymbol{x} + d$. The assignment is *invertible* if $c_j \neq 0$, on the other hand the assignment is non-invertible or *destructive* if $c_j = 0$. An invertible assignment can be handled using a substitution $\psi : \ \varphi[x_j \mapsto \frac{1}{c_j}(x_j - (\mathsf{e} - c_j x_j))]$. A destructive update is handled by first using the projection algorithm to compute $\varphi' : \ \exists x_j \ \varphi$ and then computing the intersection $\psi : \ \alpha(\varphi' \wedge x_j = e)$ using the abstraction algorithm.

*Widening.* An instance of widening consists of two SRCs $\varphi_1, \varphi_2$ such that $\varphi_1 \models \varphi_2$. Using standard widening [9], we simply drop each constraint in $\varphi_1$ that is not entailed by $\varphi_2$. Let $x_j \leq \mathsf{u}_j$ be an upper bound in $\varphi_1$. We first compute $c_j = \mathsf{max.} \ \ (x_j - \mathsf{u}_j) \ \mathsf{s.t.} \ \ \varphi_2$. If $c_j > 0$ then $\varphi_2 \not\models_W x_j \leq \mathsf{u}_j$. Therefore, we need to drop the constraint. This may be done by replacing the bound $\mathsf{u}_j$ with $\infty$. A better widening operator is obtained by first replacing each occurrence of $x_j^-$ ($x_j$ occurring with negative polarity) by a matching pair before replacing $\mathsf{u}_j$. Lower bounds such as $x_j \geq \mathsf{l}_j$ are handled symmetrically.

**Lemma 7.** *The* SRC *widening $\nabla_R$ satisfies (a) $\varphi_1, \varphi_2 \models_W \varphi_1 \nabla_R \varphi_2$; (b) any ascending chain eventually converges (even if $\models_W$ is used to detect convergence), i.e., for any sequence $\psi_1, \ldots, \psi_n, \ldots$, the widened sequence $\varphi_1, \ldots,$ satisfies $\varphi_{N+1} \models_W \varphi_N$, for some $N > 0$.*

*Narrowing.* The SRC narrowing is similar to the interval narrowing on Cousot et al. [10]. Let $\varphi_2 \models \varphi_1$. The narrowing $\varphi_1 \bigtriangleup_{\mathrm{R}} \varphi_2$ is given by replacing every $\pm\infty$ bound in $\varphi_1$ by the corresponding bound in $\varphi_2$.

**Lemma 8.** *For any* SRC*s $\varphi_1$ and $\varphi_2$, s.t. $\varphi_2 \models \varphi_1$, $\varphi_1 \bigtriangleup_{\mathrm{R}} \varphi_2 \models_W \varphi_1$. Furthermore, the narrowing iteration for* SRC *domain converges.*

*Equalities.* While equalities can be captured in the SRC domain itself, it is beneficial to compute the equality constraints separately. An equality constraint can be stored as $A\boldsymbol{x} + \boldsymbol{b} = 0$ where $A$ is a $n \times n$ matrix. In practice, we store $A$ in its triangulated form assuming some ordering on the variables. Therefore, it is possible to construct the product domain of SRC and linear equalities wherein both domains share the same variable ordering. The equality part is propagated using Karr's analysis [19].

Using the same variable ordering allows us to share information between the two domains. For instance, $\pm\infty$ bounds for the SRC component can be replaced with bounds inferred from the equality constraints during the course of the analysis. The equality invariants can also be used to delay widening. Following the polyhedral widening operator of Bagnara et al., we do not apply widening if the equality part has decreased in rank during the iteration [1].

### Variable Ordering

We now consider the choice of the variable ordering. The variable ordering used in the analysis has a considerable impact on its precision. The ideal choice of a variable ordering requires us to assign the higher indices to variables which are likely to be unbounded, or have constant bounds. Secondly, if a variable $x$ is defined in terms of $y$ in the program flow, it is more natural to express the bounds of $x$ in terms of $y$ than the other way around. We therefore consider two factors in choosing a variable ordering: (a) ordering based on variable type or its purpose in the code; and (b) ordering based on variable dependencies.

The determination of the "type" or "purpose" of a variable is made using syntactic templates. For instance, variables used as loop counters, or array indices are assigned lower indices than loop bounds or those that track array/pointer lengths. Similarly, variables used as arguments to functions have higher indices than local variables inside functions. These variables are identified in the front end during CFG construction using a simple variable dependency analysis.

Variables of a similar type are ordered using data dependencies. A dataflow analysis is used to track dependencies among a variable. If the dependency information between two variables is always uni-directional we use this information to determine a variable ordering. Finally, variables which cannot be otherwise ordered in a principled way are ordered randomly.

## 5 Implementation

We have implemented an analysis tool to prove array accesses safe as part of the ongoing F-SOFT project [18]. Our analyzer is targeted towards proving numerous runtime safety properties of C programs including array and pointer access checks. The analyzer is *context sensitive*, by using call strings to track contexts. While recursive functions cannot be handled directly, they may be abstracted by unrolling to some fixed length and handling the remaining calls context insensitively. Our abstract interpreter supports a combination of different numerical domains, including constant folding, interval, octagon, polyhedron and SRC domains. For our experiments, we used off-the-shelf implementations of the octagon abstract domain library [20], and the Parma Polyhedron Library [2]. Each library was used with the same abstract interpreter to carry out the program analysis.

The tool constructs a CFG representation from the program, which is simplified using program slicing [29], constant propagation, and optionally by interval analysis. A linearization abstraction converts operations such as multiplication and integer division into non-deterministic choices. Arrays and pointers are modeled by their allocated sizes while array contents are abstracted away. Pointer aliasing is modeled soundly using a flow insensitive alias analysis.

*Variable clustering.* The analysis model size is reduced by creating small *clusters* of related variables. For each cluster, statements that involve variables not belonging to the current cluster are abstracted away. The analysis is performed on these abstractions. A property is considered proved only if it can be proved in each context by some cluster abstraction. Clusters are detected heuristically by a backward traversal of the CFG, collecting the variables that occur in the same expressions or conditions. The backward traversal is stopped as soon as the number of variables in a cluster first exceeds 20 variables for our experiments. The number of clusters ranges from a few hundreds to nearly 2000 clusters.

*Iteration Strategy.* The fixpoint computation is performed by means of an upward iteration using widening to converge to some fixed point followed by a downward iteration using narrowing to improve the fixed point until no more improvements are possible. To improve the initial fixed point, the onset of widening is delayed by a fixed number of iterations (2 iterations for our experiments). The iteration strategy used is *semi-naive*. At each step, we minimize the number of applications of post conditions by keeping track of nodes whose abstract state changed in the previous iteration. In the case of the polyhedral domain, the narrowing phase is cut off after a fixed number of iteration to avoid potential non termination.

## 6 Experiments

Our experiments involved the verification of C programs for runtime errors such as buffer overflows, null pointer accesses, and string library usage checks. The domains are compared simply based on their ability to prove properties.

**Small Benchmarks.** We first compare the domains on a collection of small example programs [24]. These programs are written in the C language, and range from 20-400 lines of code. The examples typically consist of statically or dynamically allocated arrays accessed inside loops using aliased pointers, and passed as parameters to string/standard library functions.

Table 1 summarizes the results on these examples. The table on the left shows the total running times and the number of properties established. The properties proved by the domains are compared pairwise. The pairwise comparison summarizes the number of properties that each domain could (not) prove as compared to other domains. In general, the SRC domain comes out slightly ahead in terms of proofs, while remaining competitive in terms of time. An analysis of the failed

**Table 1.** Comparison results on small examples. **Prog.**: Number of programs, **#Prp.**: total number of properties, **Prf**: number of proofs, **T**: time taken in seconds. Detailed pairwise comparison of the proofs is shown on the right.

| Prog. | #Prp. | Int. | | Oct. | | Poly. | | SRCs | |
|---|---|---|---|---|---|---|---|---|---|
| | | Prf | T | Prf | T | Prf | T | Prf | T |
| 48 | 480 | 316 | 9 | 340 | 29 | 356 | 413 | 360 | 81 |

| | vs. Int. | vs. Oct. | vs. SRC |
|---|---|---|---|
| Oct. | $+29/-5$ | | |
| SRC | $+45/-1$ | $+23/-3$ | |
| Poly | $+46/-6$ | $+24/-8$ | $+7/-11$ |

**Table 2.** Comparison of different implementation choices for SRC. **SIMP**: simplex instead of weak optimization, **Random**: Random var. ordering, **Reverse**: reversal of the implemented var. ordering, **Random**: Resolve conflicts randomly, **Lex:** choose expr. with lower lex order, **Arg1**: Always retain, **Arg2**: Always replace existing expr.

| opt. | | Var. Ordering | | | | Conflict Resolution | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SIMP | | Random | | Reverse | | Random | | Lex | | Arg1 | | Arg2 | |
| Prf | T | Prf | T | Prf | T | Prf | T | Prf | T | Prf | T | Prf | T |
| 0/0 | 906 | +4/-23 | 114 | +1/-29 | 132 | +1/-13 | 80 | +2/-4 | 81 | +6/-8 | 80 | +6/-8 | 68 |

proofs revealed that roughly 25 are due to actual bugs (mostly unintentional) in the programs, while the remaining were mostly due to modeling limitations.

**Comparison of Implementation Choices.** Our implementation of SRCs requires heuristics for optimization, variable ordering and conflict resolution while abstracting. Table 2 compares the proofs and running times for some alternative strategies for these operations. Each experiment in the table changes one option at a time, leaving the others unchanged. The choices we made for these strategies perform better than the more ad-hoc strategies used in these experiments. In particular, the difference is most pronounced when the variable ordering used is exactly the reverse of that suggested by our heuristic.

**Network Controller Study.** We studied the performance of our analyzer on a commercial network controller implementation. The analysis is started at different root functions assuming an unknown calling environment. Root functions are chosen based on their position in the global call graph. Each analysis run first simplifies the model using slicing, constant folding and interval analysis. Table 3 shows each of these functions along with the number of properties sliced away as a result of all the front-end simplifications. Also note that a large fraction of the properties can be handled simply by using interval analysis and constant folding. Slicing the CFG to remove these properties triggers a large reduction in the CFG size.

Table 4 compares the performance of the SRC domain with the octagon and polyhedral domains on the CFG simplified by slicing, constant folding and intervals. The interval domain captures many of the easy properties including the common case of static arrays accessed in loops with known bounds. While the SRC and octagon domains can complete on all the examples even in the absence

**Table 3.** Front end statistics for network controller. **#BB**: number of basic blocks, **#Fun**: number of functions, **#BC**: $\sum_{\mathsf{block}\ n} \#Contexts(n)$, # of CFG blocks weighted by the # of contexts for each block , **#Prop**: number of properties, **Proof**: number of proofs by constant folding + intervals, **Time**: simplification time (sec), **#BC_Simpl**: Block-contexts after simplification.

| Name | KLOC | #BB | #Fun | #BC | #Prop | Simplifications | | |
|---|---|---|---|---|---|---|---|---|
| | | $\times 10^3$ | | $\times 10^3$ | | Proof | Time | #BC_Simpl |
| F1 | 5.9 | 1.6 | 11 | 2.0 | 441 | 208 | 24 | 1.6 |
| F2 | 6.4 | 1.7 | 9 | 2.2 | 545 | 223 | 77 | 1.9 |
| F3 | 7.2 | 2.1 | 11 | 2.6 | 613 | 424 | 58 | 1.5 |
| F4 | 9.4 | 3.3 | 12 | 4.8 | 995 | 859 | 128 | 1.6 |
| F5 | 11.3 | 3.8 | 16 | 4.5 | 1133 | 644 | 268 | 3.2 |
| F6 | 15.0 | 5.3 | 15 | 10.0 | 1611 | 1427 | 451 | 2.1 |
| F7 | 14.5 | 2.1 | 5 | 2.5 | 733 | 354 | 30 | 1.5 |
| F8 | 25.7 | 9.0 | 5 | 29.6 | 2675 | 2641 | 1266 | 2.4 |
| F9 | 23.0 | 8.1 | 8 | 11.9 | 2461 | 2391 | 1350 | 2.0 |
| F10 | 45.4 | 16.6 | 59 | 60.6 | 4671 | 4627 | 2h30m | 6.6 |
| 10 | 164 | | | | 15878 | 13798 | 12850 | |

**Table 4.** Comparing the performance of abstract domains on simplified CFG. $P_{Oct}$: number of octagon proofs, $T_{Oct}$: octagon analysis time (seconds), $P_{SRC}, T_{SRC}$: SRC proof and time taken, $P_{Poly}, T_{Poly}$: polyhedron time and proof.

| Function | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | F10 | Tot |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_{Oct}$ | 56 | 0 | 23 | 56 | 146 | 56 | 28 | 14 | 0 | 0 | 379 |
| $T_{Oct}$ | 11 | 30.7 | 9 | 3.2 | 105 | 7.7 | 10.4 | 1.3 | .9 | .4 | 180 |
| $P_{SRC}$ | 56 | **12** | 22 | 56 | 146 | 56 | 28 | 14 | 0 | **14** | **404** |
| $T_{SRC}$ | 18.2 | 59.1 | 21.0 | 7.6 | 291.7 | 17 | 20.7 | 0.7 | 1.5 | 0.5 | 439 |
| $P_{Poly}$ | 42 | 0 | 23 | 0 | 62 | 0 | 0 | 0 | 0 | 0 | 127 |
| $T_{Poly}$ | 63 | 684 | 75 | 29 | 1697 | 63.5 | 51.1 | 2.7 | 4.2 | 1.4 | 2672 |

of such simplifications, running interval analysis as a pre-processing step nevertheless lets us focus on those properties for which domains such as octagons, SRC and polyhedra are really needed. In many situations, the domains produce a similar bottom line. Nevertheless, there are cases where SRCs capture proofs missed by octagons and polyhedra. The SRC domain takes roughly 2.5× more time than the octagon domain. On the other hand, the polyhedral domain proves much fewer properties than both octagons and SRCs in this experiment, while requiring significantly more time. We believe that the iteration strategy used, especially the fast onset of widening and the narrowing cutoff for polyhedra may account for the discrepancy. On the other hand, increasing either parameter only serve to slow the analysis down further. In general, precise widening operators [1] along with techniques such as *lookahed widening* [16], *landmark-based* *widening* [26] or widening with *acceleration* [15] can compensate for the lack of a good polyhedral narrowing.

## 7 Conclusion

We have presented an abstract domain using symbolic ranges that captures many properties that are missed by other domains such as octagons and intervals. At the same time, our domain does not incur the large time complexity of the polyhedral domain. In practice, we hope to use the SRC domain in conjunction with intervals, octagons and polyhedra to prove more properties with a reasonable time overhead.

Many interesting avenues of future research suggest themselves. One interesting possibility is to allow for a conjunction of many SRC constraints, each using a different variable ordering. Apart from checking overflows, the SRC domain may also be useful for analyzing the numerical stability of floating point loops [17]. The constraint handling techniques presented in this paper can be directly applied to practical tools such as ARCHER [31] and ESP [12].

**Acknowledgments.** We gratefully acknowledge Ilya Shlyakhter for his useful insights and the anonymous reviewers for their comments. We acknowledge the efforts of Antoine Miné for his Octagon library and the Parma Polyhedra library team [2] for making our experiments possible.

## References

1. Bagnara, R., Hill, P.M., Ricci, E., Zaffanella, E.: Precise widening operators for convex polyhedra. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 337–354. Springer, Heidelberg (2003)
2. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.M.: Possibly not closed convex polyhedra and the Parma Polyhedra Library. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 213–229. Springer, Heidelberg (2002)
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. . In: ACM SIGPLAN PLDI'03, vol. 548030, pp. 196–207. ACM Press, New York (2003)
4. Blume, W., Eigenmann, R.: Symbolic range propagation. In: Proceedings of the 9th International Parallel Processing Symposium (April 1995)
5. Chvátal, V.: Linear Programming. Freeman (1983)
6. Clarisó, R., Cortadella, J.: The octahedron abstract domain. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 312–327. Springer, Heidelberg (2004)
7. Costan, A., Gaubert, S., Goubault, E., Martel, M., Putot, S.: A policy iteration algorithm for computing fixed points in static analysis of programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 462–475. Springer, Heidelberg (2005)
8. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming, Dunod, pp. 106–130 (1976)

9. Cousot, P., Cousot, R.: Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM Principles of Programming Languages, pp. 238–252 (1977)
10. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to Abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
11. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among the variables of a program. In: ACM POPL, pp. 84–97. ACM, New York (1978)
12. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: Proceedings of Programming Language Design and Implementation (PLDI 2002), pp. 57–68. ACM Press, New York, NY, USA (2002)
13. Dor, N., Rodeh, M., Sagiv, M.: CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In: Proc. PLDI'03, ACM Press, New York (2003)
14. Gawlitza, T., Seidl, H.: Precise fixpoint computation through strategy iteration. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 284–289. Springer, Heidelberg (2007)
15. Gonnord, L., Halbwachs, N.: Combining widening and acceleration in linear relation analysis. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 144–160. Springer, Heidelberg (2006)
16. Gopan, D., Reps, T.W.: Lookahead widening. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 452–466. Springer, Heidelberg (2006)
17. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 18–34. Springer, Heidelberg (2006)
18. Ivančić, F., Shlyakhter, I., Gupta, A., Ganai, M. K., Kahlon, V., Wang, C., Yang, Z.: Model checking C programs using f-soft. In: ICCD, pp. 297–308 (2005)
19. Karr, M.: Affine relationships among variables of a program. Acta Inf. 6 , 133–151 (1976)
20. Miné, A.: Octagon abstract domain library, `http://www.di.ens.fr/~mine/oct/`
21. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001)
22. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
23. Rugina, R., Rinard, M.: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In: Proc. Programming Language Design and Implementation (PLDI'03), ACM Press, New York (2000)
24. Sankaranarayanan, S. NEC C language static analysis benchmarks. Available by request from, `srirams@nec-labs.com`
25. Sankaranarayanan, S., Colón, M., Sipma, H.B., Manna, Z.: Efficient strongly relational polyhedral analysis. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, Springer, Heidelberg (2006)
26. Simon, A., King, A.: Widening Polyhedra with Landmarks. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 166–182. Springer, Heidelberg (2006)
27. Simon, A., King, A., Howe, J.M.: Two Variables per Linear Inequality as an Abstract Domain. In: Leuschel, M.A. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 71–89. Springer, Heidelberg (2003)
28. Su, Z., Wagner, D.: A class of polynomially solvable range constraints for interval analysis without widenings. Theor. Comput. Sci. 345(1), 122–138 (2005)
29. Tip, F.: A survey of program slicing techniques. J. Progr. Lang. 3(3) (1995)

30. Wagner, D., Foster, J., Brewer, E., Aiken, A.: A first step towards automated detection of buffer overrun vulnerabilities. In: Proc. Network and Distributed Systems Security Conference, pp. 3–17. ACM Press, New York (2000)
31. Xie, Y., Chou, A., Engler, D.: ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. SIGSOFT Softw. Eng. Notes 28(5) (2003)
32. Zaks, A., Cadambi, S., Shlyakhter, I., Ivančić, F., Ganai, M. K., Gupta, A., Ashar, P.: Range analysis for software verification. In: Proc. Workshop on Software Validation and Verification (SVV) (2006)

# Shape Analysis with Structural Invariant Checkers*

Bor-Yuh Evan Chang[1], Xavier Rival[1,2], and George C. Necula[1]

[1] University of California, Berkeley, California, USA
[2] École Normale Supérieure, Paris, France
{bec,rival,necula}@cs.berkeley.edu

**Abstract.** Developer-supplied data structure specifications are important to shape analyses, as they tell the analysis what information should be tracked in order to obtain the desired shape invariants. We observe that data structure checking code (e.g., used in testing or dynamic analysis) provides shape information that can also be used in static analysis. In this paper, we propose a lightweight, automatic shape analysis based on these developer-supplied structural invariant checkers. In particular, we set up a parametric abstract domain, which is instantiated with such checker specifications to summarize memory regions using both notions of complete and partial checker evaluations. The analysis then automatically derives a strategy for canonicalizing or weakening shape invariants.

## 1 Introduction

Pointer manipulation is fundamental in almost all software developed in imperative programming languages today. For this reason, verifying properties of interest to the developer or checking the pre-conditions for certain complex program transformations (e.g., refactorings) often requires detailed aliasing and structural information. Shape analyses are unique in that they can provide this detailed must-alias and shape information that is useful for many higher-level analyses (e.g., typestate or resource usage analyses, race detection for concurrent programs). Unfortunately, because of precision requirements, shape analyses have been generally prohibitively expensive to use in practice.

The design of our shape analysis is guided by the desire to keep the abstraction close to informal developer reasoning and to maintain a reasonable level of interaction with the user in order to avoid excessive case analysis. In this paper, we propose a shape analysis guided by the developer through programmer-supplied data structure invariants. The novel aspect of our proposal is that these specifications are given as checking code, that is, code that could be used to verify instances dynamically. In this paper, we make the following contributions:

* This research was supported in part by the National Science Foundation under grants CCR-0326577, CCF-0524784, and CNS-0509544; and an NSF Graduate Research Fellowship.

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 384–401, 2007.

- We observe that invariant checking code can help guide a shape analysis and provides a familiar mechanism for the developer to supply information to the analysis tool. Intuitively, checkers can be viewed as programmer-supplied summaries of heap regions bundled with a usage pattern for such regions.
- We develop a shape analysis based on programmer-supplied invariant checkers (utilizing the framework of separation logic [17]).
- We introduce a notion of partial checker runs (using $-\!\!*$ ) as part of the abstraction in order to generalize programmer-supplied summaries when the data structure invariant holds only partially (Sect. 3).
- We notice that the iteration history of the analysis can be used to guide the weakening of shape invariants, which perhaps could apply to other shape analyses. We develop an automatic widening strategy for our abstraction based on this observation (Sect. 4.2).

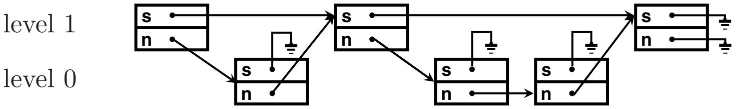
In this paper, we consider structural invariants, that is, invariants concerning the pointer structure (e.g., acyclic list, cyclic list, tree) but not data properties (e.g., orderedness). In the next section, we motivate the design of our shape analysis and highlight the challenges through an example.

## 2 Overview

In Fig. 1, we present an example analysis that checks a skip list [16] rebalancing operation to verify that it preserves the skip list structure. At the top, we show the structure of a two-level skip list. In such a skip list, each node is either level 1 or level 0. All nodes are linked together with the next field (`n`), while the level 1 nodes are additionally linked with the skip field (`s`). A level 0 node has its `s` field set to `null`. In the middle left, we give the C type declaration of a `SkipNode` and in the middle right, we give a checking routine `skip1` that when viewed as C code (assumed type safe) either diverges if there is a cycle in the reachable nodes, returns false, or returns true when the nodes reachable from the argument `l` are arranged in a skip list structure. The `skip0` function is a helper function for checking a segment of level 0 nodes. Intuitively, `skip1` and `skip0` simply give the inductive structure of skip lists.

In the bottom section of Fig. 1, we present an analysis of the rebalancing routine (`rebalance`). The **assert** at the top ensures that `skip1(l)` holds (i.e., `l` is a skip list), and the **assert** at the bottom checks that `l` is again a skip list on return. We have made explicit these pre- and post-conditions here, but we can imagine a system that connects the checker to the type and verifies that the structure invariants are preserved at function or module boundaries. In the figure, we show the abstract memory state of the analysis at a number of program points using a graphical notation, which for now, we can consider as informal sketches a developer might draw to check the code by hand. For the program points inside the loop there are two memory states shown: one for the first iteration (left) and one for the fixed point (right).

A programmer-defined checker can be used in static analysis by viewing the memory addresses it would dereference during a successful execution as describing a class of memory regions arranged according to particular constraints. We

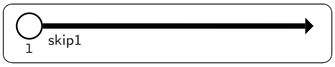


```
typedef
  struct SkipNode {
    int d;
    struct SkipNode* s;
    struct SkipNode* n;
  }
  SkipNode;
```

```
bool skip1(SkipNode* l) {
  if (l == null) return true;
  else return skip1(l->s) &&
              skip0(l->n, l->s);
}
bool skip0(SkipNode* l, SkipNode* e) {
  if (l == e) return true;
  else return l != null && l->s == null &&
              skip0(l->n, e);
}
```

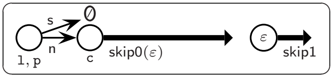
```
  void rebalance(SkipNode* l) {
    SkipNode *p, *c;
    assert (l != null && skip1(l));
```

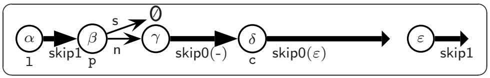
1



```
    p = l;       // previous level 1 node
2   c = l->n;    // cursor
3   l->s = null;
4   while (c != null) {
```

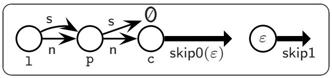
5

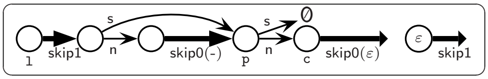




```
      if (c should be a level 1 node) {
6       p->s = c;    // set the skip pointer of the previous level 1 node
7       p = p->s;
8       c->s = null; c = c->n;
```

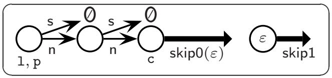
9

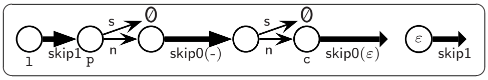




```
      }
      else {
10      c->s = null; c = c->n;
```

11





```
      }
    }
12  assert (l != null && skip1(l));
  }
```

First Iteration | At Fixed Point

**Fig. 1.** Analysis of a skip list rebalancing

build an abstraction around this summarization mechanism. To name heap objects, the analysis introduces *symbolic values* (i.e., fresh existential variables). To distinguish them from program variables, we use lowercase Greek letters ($\alpha, \beta, \gamma, \delta, \varepsilon, \pi, \rho, \ldots$). A graph node denotes a value (e.g., a memory address) and, when necessary, is labeled by a symbolic value; the 0 nodes represent null. We write a program variable (e.g., l) below a node to indicate that the value of that variable is that node. Each edge corresponds to a memory region. A thin edge denotes a *points-to* relationship, that is, a memory cell whose address is the source node and whose value is the destination node (e.g., on line 5 in the left graph, the edge labeled by n says that l->n points to c). A thick edge summarizes a memory region, i.e., some number of points-to edges. Thick edges, or *checker edges*, are labeled by a checker instantiation that describes the structure of the summarized region. There are two kinds of checker edges: complete checker edges, which have only a source node, and partial checker edges, which have both a source and a target node. Complete checker edges indicate a memory region that satisfies a particular checker (e.g., on line 1, the complete checker edge labeled skip1 says there is a memory region from l that satisfies checker skip1). Partial checker edges are generalization that we introduce in our abstraction to describe memory states at intermediate program points, which we discuss further in Sect. 3. An important point is that two distinct edges in the graph denote disjoint memory regions.

To reflect memory updates in the graph, we simply modify the appropriate points-to edges (performing strong updates). For example, consider the transition from program point 5 to point 9 and the updates on lines 6 and 7. For the updates on line 8, observe that we do not have nodes for c->s or c->n in the graph at program point 5. However, we have that from c, an instance of skip0 holds, which can be *unfolded* to materialize points-to edges for c->s and c->n (that is, conceptually unfolding one step of its computation). The update can then be reflected after unfolding.

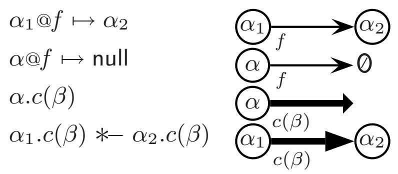
As exemplified here, we want the work performed by our shape analysis to be close to the informal, on-paper verification that might be done by the developer. The abstractions used to summarize memory regions is developer-guided through the checker specifications. While it may be reasonable to build in generic summarization strategies for common structures, like lists and trees (cf., [6,9]), it seems unlikely such strategies will suffice for other structures, like the skip lists in this example. Traversal code for checking seems like a useful and intuitive specification mechanism, as such code could be used in testing or dynamic analysis (cf., [18]).

From this example, we make some observations that guide the design of our analysis and highlight the challenges. First, in our diagrams, we have implicitly assumed a disjointness property between the regions described by edges (to perform strong updates on points-to edges). This assumption is made explicit by utilizing separation logic to formalize these diagrams (see Sect. 3). This choice also imposes restrictions on the checkers. That is, all conjunctions are separating conjunctions; in terms of dynamic checking, a compilation of skip1 must check that each address is dereferenced at most once during the traversal. Second, as with many data structure operations, the rebalance routine requires a traversal using a *cursor* (e.g., `c`). To check properties of such operations, we are often required to track information in detail locally around the cursor, but we may be able to summarize the rest rather coarsely. This summarization cannot be only for the suffix (yet to be visited by the cursor) but must also be for the prefix (already visited by the cursor) (see Sect. 3). Third, similar to other shape analyses, a central challenge is to *fold* the graphs sufficiently in order to find a fixed point (and to be efficient) while retaining enough precision. With arbitrary data structure specifications, it becomes particularly difficult. The key observation we make is that previous iterates are generally more abstract and can be used to guide the folding process (see Sect. 4.2).

## 3 Memory Abstraction

$$\begin{aligned} M ::=\ & \beta @ f \mapsto r \\ & \mid M_1 * M_2 \\ & \mid \mathsf{emp} \\ & \mid \alpha.c(\beta) \\ & \mid \alpha_1.c(\beta) \ast\!\!- \alpha_2.c(\beta) \end{aligned}$$

(a) Abstract memory states


$\alpha_1 @ f \mapsto \alpha_2$
$\alpha_1$
$f$
$\alpha_2$
$\alpha @ f \mapsto \mathsf{null}$
$\alpha$
$f$
$\emptyset$
$\alpha.c(\beta)$
$\alpha$
$c(\beta)$
$\alpha_1.c(\beta) \ast\!\!- \alpha_2.c(\beta)$
$\alpha_1$
$\alpha_2$
$c(\beta)$

(b) Edges

We describe our analysis within the framework of abstract interpretation [5]. Our analysis state is composed of an abstract memory state (in the form of a shape graph) and a pure state to track disequalities (the non-points-to constraints). We describe the memory state in a manner based largely on separation logic, so we use a notation that is borrowed from there. A memory state $M$ includes the points-to relation ($\beta @ f \mapsto r$), the separating conjunction ($M_1 * M_2$), and the empty memory state ($\mathsf{emp}$) from separation logic, which together can describe a set of possible memories that have a finite number of points-to relationships. The separating conjunction $M_1 * M_2$ describes a memory that can be divided into two disjoint regions (i.e., with disjoint domains) described by $M_1$ and $M_2$. A field offset expression $\beta @ f$ corresponds to the base address $\beta$ plus the offset of field $f$ (i.e., `&(b.f)` in C). For simplicity, we assume that all pointers occur as fields in a `struct`. R-values $r$ are symbolic expressions representing the contents of memory cells (whose precise form is unimportant but does include $\mathsf{null}$). Memory regions are summarized with applications of user-supplied checkers. We write $\alpha.c(\beta)$ to mean checker $c$ applied to $\alpha$ and $\beta$ holds (i.e., $c$ succeeds when applied to $\alpha$ and $\beta$). For example, $\alpha.\mathsf{skip1}()$ says that the `skip1` checker is successful when applied to $\alpha$. We use this object-oriented style notation to distinguish the main traversal argument $\alpha$ from any additional parameters $\beta$. These additional parameters may be used to specify additional constraints (as in the `skip0` checker in Fig. 1), but we do not traverse from them. We also introduce a notion of a *partial checker run* $\alpha_1.c(\beta) \ast\!\!- \alpha_2.c(\beta)$ that describes a memory region summarized by a segment from $\alpha_1$ to $\alpha_2$, which will be described further in the subsections below. Visually, we regard a memory state as a directed graph. The edges correspond to formulas as shown in the inset (b).[1] Each edge in a graph is considered separately conjoined (i.e., each edge corresponds to a disjoint region of memory).

**Inductive Structure Checkers.** The abstract domain provides generic support for inductive structures through *user-specified checkers*. Observe that a dynamic run of a checker, such as `skip1` (in Fig. 1), visits a region of memory starting from some root pointer, and furthermore, a successful, terminating run of a checker indicates how the user intends to access that region of memory. In the context of our analysis, a checker gives a corresponding inductively-defined predicate in separation logic and a successful, terminating run of the checker bears witness to a derivation of that predicate.

The definition of a checker $c$, with formals $\pi$ and $\rho$, consists of a finite disjunction of rules. A rule is the conjunction of a separating conjunction of a series of points-to relations and checker applications $M$ and a pure, first-order predicate $P$, written $\langle M \,;\, P\rangle$. Free variables in the rules are considered as existential variables bound at the definition. Because we view checkers as executable code, the kinds of inductive predicates are restricted. More precisely, we have the following restrictions on the $M_i$'s: (1) they do not contain partial checker applications (i.e., $*\!-$) and (2) the points-to edges correspond to finite access paths from $\pi$. In other words, each $M_i$ can only correspond to a memory region reachable from $\pi$. A checker cannot, for example, posit the existence of some pointer that points to $\pi$.

$$\boxed{\pi.c(\rho) \;:=\; \langle M_1 \,;\, P_1\rangle \vee \cdots \vee \langle M_n \,;\, P_n\rangle}$$

Each rule specifies one way to prove that a structure satisfies the checker definition, by checking that the corresponding first-order predicate holds and that the store can be separated into a series of stores, which respectively allow proving each of the separating conjuncts. Base cases are rules with no checker applications.

*Example 1 (A binary tree checker).* A binary tree with fields $\mathsf{lt}$ and $\mathsf{rt}$ can be described by a checker with two rules:

$$\pi.\mathsf{tree}() := \langle \mathsf{emp} \,;\, \pi = \mathsf{null}\rangle \vee \langle (\pi @ \mathsf{lt} \mapsto \gamma) * (\pi @ \mathsf{rt} \mapsto \delta) * \gamma.\mathsf{tree}() * \delta.\mathsf{tree}() \,;\, \pi \neq \mathsf{null}\rangle$$

*Example 2 (A skip list checker).* The “C-like” checkers for the two-level skip list in Fig. 1 would be translated to the following:

$$\begin{aligned}
\pi.\mathsf{skip1}() \;&:= \langle \mathsf{emp} \,;\, \pi = \mathsf{null}\rangle \\
&\quad\vee \langle (\pi @ \mathsf{s} \mapsto \gamma) * (\pi @ \mathsf{n} \mapsto \delta) * \gamma.\mathsf{skip1}() * \delta.\mathsf{skip0}(\gamma) \,;\, \pi \neq \mathsf{null}\rangle \\
\pi.\mathsf{skip0}(\rho) &:= \langle \mathsf{emp} \,;\, \pi = \rho\rangle \\
&\quad\vee \langle (\pi @ \mathsf{s} \mapsto \mathsf{null}) * (\pi @ \mathsf{n} \mapsto \gamma) * \gamma.\mathsf{skip0}(\rho) \,;\, \pi \neq \rho \wedge \pi \neq \mathsf{null}\rangle
\end{aligned}$$

**Segments and Partial Checker Runs.** In the above, we have built some intuition on how user-specified checkers can be utilized to give precise summaries

[1] For presentation, we show the most common kinds of edges. In the implementation, we support field offsets in most places to handle, for example, pointer to fields.

of memory regions. Unfortunately, the inductive predicates obtained from typical checkers, such as `tree` or `skip1`, are usually not general enough to capture the invariants of interest at all program points. To see this, consider the invariant at fixed point on line 5 (i.e., the loop invariant) in the skip list example (Fig. 1). Here, we must track some information in detail around a cursor (e.g., `p` and `c`), while we need to summarize *both* the already explored prefix before the cursor and the yet to be explored suffix after the cursor. Such a situation is typical when analyzing a traversal algorithm. The suffix can be summarized by a checker application $\delta.\mathsf{skip0}(\varepsilon)$ (i.e., the skip0 edge from `c`), but unfortunately, the prefix segment (i.e., the region between `l` and `p`) cannot.

Rather than require more general checker specifications sufficient to capture these intermediate invariants, we introduce a generic mechanism for summarizing prefix segments. We make the observation that they are captured by *partial checker runs*. In terms of inductively-defined predicates, we want to consider partial derivations, that is, derivations with a hole in a subtree. This concept is internalized in the logic with the separating implication. For example, the segment from `l` to `p` on line 5 corresponds to the partial checker application $\alpha.\mathsf{skip1}() \ast\!- \beta.\mathsf{skip1}()$. Informally, a memory region satisfies $\alpha.\mathsf{skip1}() \ast\!- \beta.\mathsf{skip1}()$ if and only if for any disjoint region that satisfies $\beta.\mathsf{skip1}()$ (i.e., is a skip list from $\beta$), then conjoining that region satisfies $\alpha.\mathsf{skip1}()$ (i.e., makes a complete skip list from $\alpha$). This statement entails that $\beta$ is reachable from $\alpha$. Our notation for separating implication is reversed compared to the traditional notation $-\!\ast$ to mirror more closely the graphical diagrams. Our use of separating implication is restricted to the form where the premise and conclusion are checker applications that differ only in the unfolding argument because these are the only partial checker edges our analysis generates.

**Semantics of Shape Graphs.** The *concretization* of an abstract memory state with checkers is defined by induction on the structure of such memory states and on the unfolding of inductive checkers with the usual semantics of the separation logic connectors. A concrete store $\sigma$ is part of the concretization of an abstract memory state $M$ if and only if there exists a mapping of the abstract nodes in $M$ into concrete addresses in $\sigma$ (a valuation), such that $M$ under the valuation is satisfied by $\sigma$. Further details, including a full definition, is given in the extended version [3].

## 4 Analysis Algorithm

In this section, we describe our shape analysis algorithm. Like many other shape analyses, we have a notion of *materialization*, which reifies memory regions in order to track updates, as well as *blurring* or *weakening*, which (re-)summarizes certain memory regions in order to obtain a terminating analysis. For us, we materialize by *unfolding* checker edges (Sect. 4.1) and weaken by *folding* memory regions back into checker edges (Sect. 4.2). Like others, we materialize as needed to reflect updates and dereferences, but instead of weakening eagerly, we delay weakening in order to use history information to guide the process.

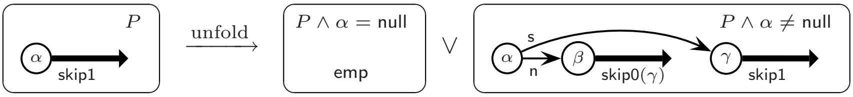
Our shape analysis is a standard forward analysis that computes an abstract state at each program point. In addition to the memory state (as described in Sect. 3), the analysis also keeps track of a number of pure constraints $P$ (pointer equalities and disequalities). Furthermore, we maintain some disjunction, so our analysis state has essentially the following form: $\langle M_1 \,;\, P_1\rangle \vee \langle M_2 \,;\, P_2\rangle \vee \cdots \vee \langle M_n \,;\, P_n\rangle$ (for unfoldings and acyclic paths where needed). Additionally, we keep the values of the program variables (i.e., the stack frame) in an abstract environment $E$ that maps program variables to symbolic values that denote their contents.[2]

### 4.1 Abstract Transition and Checker Unfolding

Because each edge in the graph denotes a separate memory region, the atomic operations (i.e., mutation, allocation, and deallocation) are straightforward and only affect graphs locally. As alluded to in Sect. 2, mutation reduces to the flipping of an edge when each memory cell accessed in the statement exists in the graph as a points-to edge. This strong update is sound because of separation (that is, because each edge is a disjoint region).

When there is no points-to edge corresponding to a dereferenced location because it is summarized as part of a checker edge, we first materialize points-to edges by *unfolding* the checker definition (i.e., conceptually unfolding one-step of the checker run). We unfold only as needed to expose the points-to edge that corresponds to the dereferenced location. Unfolding generates one graph per checker rule, obtained by replacing the checker edge with the points-to edges and the recursive checker applications specified by the rule; the pure constraints in the rule are also added to pure state. In case we derive a contradiction (in the pure constraints), then those unfolded elements are dropped. Though, unfolding may generate a *disjunction* of several graphs. A fundamental property of unfolding is that the join of the concretizations of the resulting graphs is equal to the concretization of the initial graph.

*Example 3 (Unfolding a skip list).* We exhibit an unfolding of the skip1 checker from Example 2. The addition of the pure constraints are shown explicitly.



### 4.2 History-Guided Folding

We need a strategy to identify sub-graphs that should be *folded* into complete or partial checker edges. What kinds of sub-graphs can be summarized without losing too much precision is highly dependent on the structures in question and the code being analyzed. To see this, consider the fixed-point graph at program point 5 in this skip list example (Fig. 1). One could imagine folding the points-to

[2] In implementation, we instead include the stack frame in $M$ to enable handling address of local variable expressions (as in C) in a smooth manner.

edges corresponding to `p->n` and `p->s` into one summary region from `p` to `c` (i.e., eliminating the node labeled $\gamma$), but it is necessary to retain the information that `p` and `c` are "separated" by at least *one* `n` field. Keeping node $\gamma$ expresses this fact. Rather than using a *canonicalization* operation that looks only at one graph to identify the sub-graphs that should be summarized, our weakening strategy is based on the observation that previous iterates at loop join points can be utilized to guide the folding process. In this subsection, we define the *approximation test* and *widening* operations (standard operations in abstract interpretation-based static analysis) over graphs as a simultaneous traversal over the input graphs.

**Approximation Test.** The approximation test on memory states $M_1 \sqsubseteq M_2$ takes two graphs as input and tries to establish that the concretization of $M_1$ is contained in the concretization of $M_2$ (i.e., $M_1 \Rightarrow M_2$). Static analyses rely on the approximation test in order to ensure the termination of fixed point computation. We also utilize it to collapse extraneous disjuncts in the analysis state and most importantly, as a sub-routine in the widening operation. Roughly speaking, our approximation test checks that graph $M_1$ is equivalent to graph $M_2$ up to unfolding of $M_2$. That is, the basic idea is to determine whether $M_1 \sqsubseteq M_2$ by reducing to stronger statements either by matching edges on both sides or by unfolding $M_2$. To check this relation, we need a correspondence between nodes of $M_1$ and nodes of $M_2$. This correspondence is given by a mapping $\Phi$ from nodes of $M_2$ to those of $M_1$. The condition that $\Phi$ is such a function ensures any aliasing expressed in $M_2$ is also reflected in $M_1$. If at any point, this condition on $\Phi$ is violated, then the test fails.

*Initialization.* The mapping $\Phi$ plays an essential role in the algorithm itself since it gives the points from where we should compare the graphs. It is initialized using the environment and then extended as the input graphs are traversed. The natural starting points are the nodes that correspond to the program variables (i.e., the initial mapping $\Phi_0 = \{E_2(x) \to E_1(x) \mid x \in \mathbf{Var}\}$).

*Traversal.* After initialization, we decide the approximation relation by traversing the input graphs and attempting to match all edges. To check region disjointness (i.e., linearity), when edges are matched, they are "consumed". If the algorithm gets stuck where not all edges are "consumed", then the test fails. To describe this traversal, we define the judgment $M_1 \sqsubseteq M_2[\Phi]$ that says, "$M_1$ is approximated by $M_2$ under $\Phi$."

In the following, we describe the rules that define $M_1 \sqsubseteq M_2[\Phi]$ by following the example derivation shown in Fig. 2 (from goal to axiom). A complete listing of the rules is given in the extended version [3] (Appendix A). In Fig. 2, the top line shows the initial goal with a particular initialization for $\Phi$. Each subsequent line shows a step in the derivation (i.e., a rewriting step) that is obtained by applying the rule named on the right. The highlighting of nodes and edges indicates where the rewriting applies. We are able to prove that the left graph is approximated by the right graph because we reach $\mathsf{emp} \sqsubseteq \mathsf{emp}[\Phi]$.

First, consider the application of the pointsto rule (line 3 to 4). When both $M_1$ and $M_2$ have the same kind of edge from matched nodes, the approximation

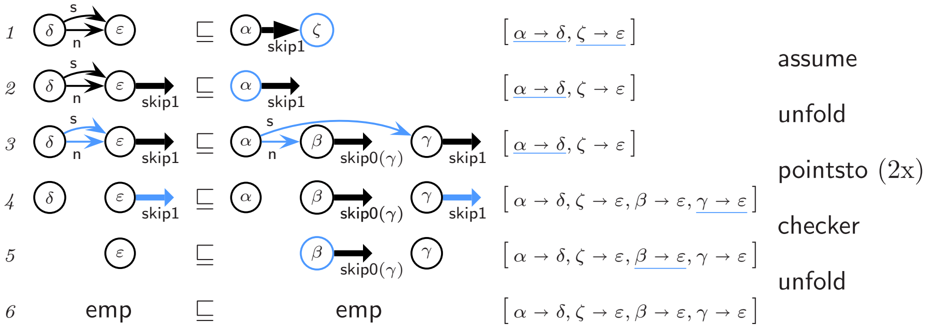


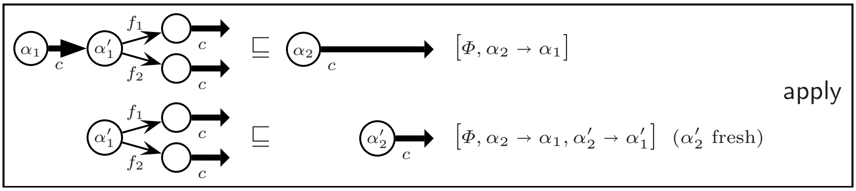
**Fig. 2.** Testing approximation by reducing to stronger statements

relation obviously holds for those edges, so those edges can be consumed. Any target nodes are then added to the mapping $\Phi$ so that the traversal can continue from those nodes. In this case, the s and n points-to edges match from the pair $\alpha \to \delta$. With this matching, the mappings $\beta \to \varepsilon, \gamma \to \varepsilon$ are added. We highlight in $\Phi$ with underlines the mappings that must match for each rule to apply. The checker rule is the analogous matching rule for complete checker edges. We apply this edge matching only to points-to edges and complete checker edges. Partial checker edges are treated separately as described below.

Partial checker edges are handled by taking the separating implication interpretation, which becomes critical here. We use the assume rule (as in the first step in Fig. 2) to reduce the handling of partial checker edges in $M_2$ to the handling of complete checker edges (i.e., a "$-\!\!*$ right" in sequent calculus or "$-\!\!*$ introduction" in natural deduction). It extends the partial checker edge in $M_2$ to a complete checker edge by adding the corresponding completion to $M_1$. A key aspect of our algorithm is that this rule only applies when we have matched both the source and target nodes of the partial checker edge, that is, we have delineated in $M_1$ the region that corresponds to the partial checker edge in $M_2$.

Now, consider the first application of unfold in Fig. 2 (line 2 to 3) where we have a complete checker edge from $\alpha$ on the right, but we do not have an edge from $\delta$ on the left that can be immediately matched with it. In this case, we unfold the complete checker edge. In general, the unfolding results in a disjunction of graphs (one for each rule, Sect. 4.1), so the overall approximation check succeeds if the approximation check succeeds for any *one* of the unfolded graphs. Note that on an unfolding, we must also remember the pure constraint $P$ from the rule, which must be conjoined to the pure state on the right when we check the approximation relation on the pure constraints. In the second application of unfold in Fig. 2 (line 5 to 6), the unfolding of $\beta$.skip0($\gamma$) is to emp because we have that $\beta = \gamma$. This equality arises because they are both unified with $\varepsilon$ (specifically, the pointsto steps added $\beta \to \varepsilon$ and $\gamma \to \varepsilon$ to $\Phi$).

Finally, we also have a rule for partial checkers in $M_1$ (i.e., a corresponding "left" or "elimination" rule). Since it is not used in the above example, we present it below schematically:



The rule is presented in the same way as in the example (i.e., with the goal on top). Conceptually, this rule can be viewed as a kind of unfolding rule where the complete checker edge in $M_2$ is unfolded the necessary number of steps to match the the partial checker edge in $M_1$.

Informally, the soundness of the approximation test can be argued from separation logic principles and from the fact that unfoldings have equivalent concretizations. The approximation test is, however, incomplete (i.e., it may fail to establish that an approximation relation between two graphs when their concretizations are ordered by subset containment). Rather these rules have been primarily designed to be effective in the way the approximation test is used by the widening operation as described in the next subsection where we need to determine if $M_1$ is an unfolded version of $M_2$.

**Widening.** In this subsection, we present an upper bound operation $M_1 \triangledown M_2$ that we use as our widening operator at loop join points. The case of disjunctions of graphs will be addressed below. At a high-level, the upper bound operation works in a similar manner as compared to the approximation test. We maintain a node pairing $\Psi$ that relates the nodes of $M_1$ and $M_2$. Because we are computing an upper bound here, the pairing $\Psi$ need not have the same restriction as in the approximation test; it may be any relation on nodes in $M_1$ and $M_2$. From this pairing, we simultaneously traverse the input graphs $M_1$ and $M_2$ consuming edges. However, for the upper bound operation, we also construct the upper bound as we consume edges from the input graphs. Intuitively, the basic edge matching rules will lay down the basic structure of the upper bound and guide us to the regions of memory that need to be folded.

*Initialization.* The initialization of $\Psi$ is the analogous to the approximation test initialization: we pair the nodes that correspond to the values of each variable from the environments (i.e., the initial pairing $\Psi_0 = \{ \langle E_1(x), E_2(x) \rangle \mid x \in \mathbf{Var} \}$).

*Traversal.* To describe the upper bound computation, we define a set of rewriting rules of the form $\Psi \mathbin{⨾} (M_1 \triangledown M_2) \circledast M \twoheadrightarrow \Psi' \mathbin{⨾} (M_1' \triangledown M_2') \circledast M'$. Initially, $M$ is $\mathsf{emp}$, and then we try to rewrite until $M_1'$ and $M_2'$ are $\mathsf{emp}$ in which case $M'$ is the upper bound. A node in $M$ corresponds to a pair (from $M_1$ and $M_2$). Conceptually, we build $M$ with nodes labeled with such pairs and then relabel each distinct pair with a distinct symbolic value at the end.

Figure 3 shows an example sequence of rewritings to compute an upper bound. A complete listing of the rewrite rules is given in the extended version [3] (Appendix B). We elide the pairing $\Psi$, as it can be read off from the nodes in the upper bound graph $M$ (the rightmost graph). The highlighting of nodes in the upper bound graph indicate the node pairings that are required to apply the

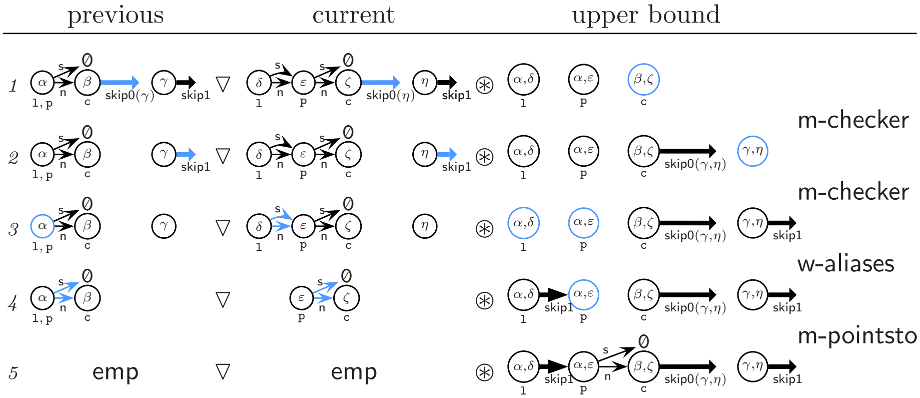
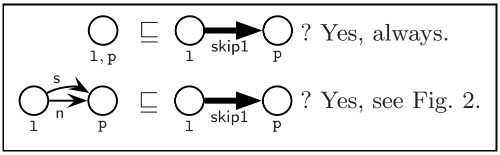


**Fig. 3.** An example of computing an upper bound. The inputs are the graphs on the first iteration at program points 5 and 9 in the skip list example (Fig. 1). The fixed-point graph at 5 is obtained by computing the upper bound of this result and the upper bound of the first-iteration graphs at 5 and 11.

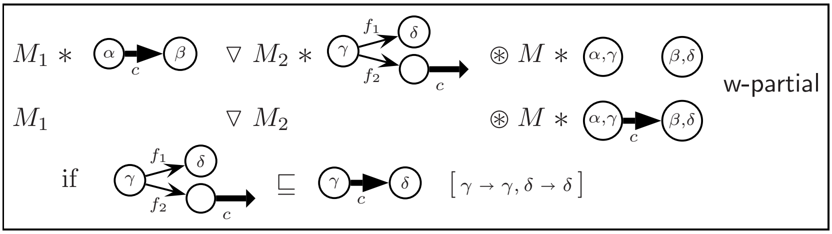
rule, and the highlighting of edges in the input graphs show which edges are consumed in the rewriting step. Roughly speaking, the upper bound operation has two kinds of rules: matching rules for when we have the same kind of edge on both sides (like in the approximation test) and weakening rules where we have identified a memory region to fold. We use the prefix m- for the matching rules and w- for the weakening rules.

Line 1 shows the state after initialization: we have nodes in upper bound graph for the program variables. The first two steps (applying rule m-checker) match complete checker edges (first from $\langle \beta, \zeta \rangle$ and then from $\langle \gamma, \eta \rangle$). Note that the second application is enabled by the first where we add the pair $\langle \gamma, \eta \rangle$. Extra parameters are essentially implicit target nodes.



The core of the upper bound operation are three weakening rules where we fold memory regions. The next rule application w-aliases is such a weakening step (line 3 to 4). In this case, a node on one side is paired with two nodes on the other ($\langle \alpha, \delta \rangle$ and $\langle \alpha, \varepsilon \rangle$). This situation arises where on one side, we have must-alias information, while the other side does not (`l` and `p` are aliased on the left but not on the right). In this case, we want to weaken both sides to a partial checker edge. To see that this is indeed an upper bound for these regions, consider the diagram in the inset. As shown on the first line, aliases can always be weakened to a partial checker edge (intuitively, from a zero-step segment to a zero-or-more step segment). On the second line, we need to check that a skip1 checker edge is indeed weaker than the region between $\delta$ and $\varepsilon$. This check is done using the approximation test described in the previous subsection. The check we need to perform here is the example shown in Fig. 2. Observe that we utilize the edge matching rules that populates $\Psi$ to delineate the region to be folded (e.g., the region between $\delta$ and $\varepsilon$ in the right graph). For the w-aliases rule, we do not specify here how the checker $c$ is determined, but in practice, we can limit the checkers that need to be tried by, for example, tracking the type of the node (or looking at the fields used in outgoing points-to edges).

There are two other weakening rules w-partial and w-checker that are not used in the above example. Rule w-partial applies when we identify that an (unfolded) memory region on one side corresponds to a partial checker edge on the other. In this case, we weaken to the partial checker edge if we can show the partial checker edge is weaker than the memory region. Rule w-partial is shown below schematically:



Observe that we find out that the region in the right graph must be folded because the corresponding region in the left graph is folded (and also indicates which checker to use). Rule w-checker is the analogous rule for a complete checker edge.

In Fig. 3, the last step is simply matching points-to edges. When we reach emp for $M_1$ and $M_2$, then $M$ is the upper bound. In general, if, in the end, there are regions we cannot match or weaken in the input graphs, we can obtain an upper bound by weakening those regions to $\top$ in the resulting graph (i.e., a summary region that cannot be unfolded). This results in an enormous loss in precision that we would like avoid but can be done if necessary.

$$\frac{M_1' \Rightarrow M \quad M_2' \Rightarrow M}{(M_1 * M_1') \vee (M_2 * M_2') \Rightarrow (M_1 \vee M_2) * M}$$

*Soundness.* The basic idea is that we compute an upper bound by rewriting based on the derived rule of inference in separation logic shown in the inset. For each memory region in the input graphs, either they have the same structure in the input graphs and we preserve that structure or we weaken to a checker edge only when we can decide the weakening with $\sqsubseteq$. That is, during the traversal, we simply alternate between weakening memory regions in each input graph to make them match and applying the distributivity of separating conjunction over disjunction to factor out matching regions.

*Termination.* We shall use this upper bound operation as our widening operator, so we check that it has the stabilizing property (i.e., successive iterates eventually stabilize) to ensure termination of the analysis. Consider an infinite ascending chain $M_0 \sqsubseteq M_1 \sqsubseteq M_2 \sqsubseteq \cdots$ and the corresponding widening chain $M_0 \sqsubseteq (M_0 \nabla M_1) \sqsubseteq ((M_0 \nabla M_1) \nabla M_2) \sqsubseteq \cdots$ (i.e., the sequence of iterates). The widening chain stabilizes because the successive iterates are bounded by the size of $M_0$. Over the sequence of iterates, the only rule that may produce additional edges not present in $M_0$ is w-aliases, but its applicability is limited by the number of nodes. Then, nodes are created in the result only in two cases: the target node when matching points-to edges (m-pointsto) and any additional parameter nodes when matching complete checker edges (m-checker). Points-to and complete checker edges are only created in the resulting graph because of matching, so the number of nodes is limited by the points-to and complete checker edges in $M_0$.

*Disjunctions of graphs.* In general, we consider widening disjunctions of graphs. The widening operator for *disjunctions* is based on the operator for graphs and attempts to find pairs that can be widened precisely in the sense that no region need be weakened to $\top$ (i.e., because an input region could not be matched). In addition to this selective widening process, the widening may leave additional disjuncts, up to some fixed limit (perhaps based on trace partitioning [11]).

### 4.3 Extensions and Limitations

The kinds of structures that can be described with our checkers are essentially trees with regular sharing patterns, which include skip lists, circular lists, doubly-linked lists, and trees with parent pointers. Intuitively, these are structures where one can write a recursive traversal that dereferences each field once (plus pointer equality and disequality constraints). However, the effectiveness of our shape analysis is not the same for all code using these structures. First, we materialize only when needed by unfolding inductive definitions, which means that code that traverse structures in a different direction than the checker are more difficult to analyze. This issue may be addressed by considering additional materialization strategies. Second, in our presentation, we consider partial checker edges with one hole (i.e., a separating implication with one premise). This formulation handles code that use cursors along a path through the structure but not code that uses multiple cursors along different branches of a structure.

## 5 Experimental Evaluation

| | Code | Analysis | Max. | Max. |
|---|---|---|---|---|
| | Size | Time | Graphs | Iter. |
| Benchmark | (loc) | (sec) | (num) | (num) |
| list reverse | 19 | 0.007 | 1 | 3 |
| list remove element | 27 | 0.016 | 4 | 6 |
| list insertion sort | 56 | 0.021 | 4 | 7 |
| binary search tree find | 23 | 0.010 | 2 | 4 |
| skip list `rebalance` | 33 | 0.087 | 6 | 7 |
| `scull` driver | 894 | 9.710 | 4 | 16 |

We evaluate our shape analysis using a prototype implementation for analyzing C code. Our analysis is written in OCaml and uses the CIL infrastructure [14]. We have applied our analysis to a number of small data structure manipulation benchmarks and a larger Linux device driver benchmark (`scull`). In the table, we show the size in pre-processed lines of code, the analysis times on a 2.16GHz Intel Core Duo with 2GB RAM, the maximum number of graphs (i.e., number of disjuncts) at any program point, and the maximum number iterations at any program point. In each case, we verified that the data structure manipulations preserved the structural invariants given by the checkers. Because we only fold into checkers based only on history information, we typically cannot generate the appropriate checker edge when a structure is being constructed. This issue could be resolved by using constructor functions with appropriate post-conditions or perhaps a one graph operation that can identify potential foldings. For these experiments, we use a few annotations that add a checker edge that say, for example, treat this `null` as the empty list (1 each in list insertion sort and skip list `rebalance`).

The `scull` driver is from the Linux 2.4 kernel and was used by McPeak and Necula [12]. The main data structure used by the driver is an array of doubly-linked lists. Because we also do not yet have support for arrays, we rewrote the array operations as linked-list operations (and ignored other `char` arrays). We analyzed each function individually by providing appropriate pre-conditions and inlining all calls, as our implementation does not yet support proper interprocedural analysis. One function (`cleanup_module`) was not completely analyzed because of an incomplete handling of the array issues; it is not included in the line count. We also had 6 annotations for adding checker edges in this example. In all the test cases (including the driver example), the number of graphs we need to maintain at any program point (i.e., the number of disjuncts) seems to stay reasonably low.

## 6 Related Work

Shape analysis has long been an active area of research with numerous algorithms proposed and systems developed. Our analysis is closest to some more recent work on separation logic-based shape analyses by Distefano *et al.* [6] and Magill *et al.* [9]. The primary difference is that a list segment abstraction is built into their analyses, while our analysis is parameterized by inductive checker definitions. To ensure termination of the analysis, they use a canonicalization operation on list segments (an operation from a memory state to a memory state), while we use a history-guided approach to identify where to fold (an operation from two memory states to one). Note that these approaches are not incompatible with each other, and they have different trade-offs. The additional history information allowed us to develop a generic weakening strategy, but because we are history-dependent, we cannot weaken whenever (e.g., we cannot weaken aggressively after each update). Recently, Berdine *et al.* [2] have developed a shape analysis over generalized doubly-linked lists. They use a higher-order list segment predicate that is parameterized by the shape of the “node”, which essentially adds a level of polymorphism to express, for example, a linked list of cyclic doubly-linked lists. We can instead describe custom structures monomorphically with the appropriate checkers, but an extension for polymorphism could be very useful.

Lee *et al.* [8] propose a shape analysis where memory regions are summarized using grammar-based descriptions that correspond to inductively-defined predicates in separation logic (like our checkers). A nice aspect of their analysis is that these descriptions are derived from the construction of the data structure (for a certain class of tree-like structures). For weakening, they use a canonicalization operation to fold memory regions into grammar-based descriptions (non-terminals), but to ensure termination of the analysis, they must fix in advance a bound on the number of nodes that can be in a canonicalized graph.

TVLA [18] is a very powerful and generic system based on three-valued logic and is probably the most widely applied tool for verifying deep properties of complex heap manipulations. The framework is parametric in that users can provide specifications (instrumentation predicates) that affect the kinds of structures tracked by the tool. Our analysis is instead parameterized by inductive checker definitions, but since we focus on structural properties, we do not handle any data invariants. Much recent work has been targeted at improving the scalability of TVLA. Manevich *et al.* [10] describe a strategy to merge memory states whose canonicalizations are "similar" (i.e., have isomorphic sets of individuals). Our folding strategy can be seen as being particularly effective when the memory states are "similar"; like them, we would like to use disjunction when the strategy is ineffective. Arnold [1] identifies an instance where a more aggressive summarization loses little precision (by allowing summary nodes to represent zero-or-more concrete nodes instead of one-or-more). Our abstraction is related in that our checker edges denote zero-or-more steps.

Hackett and Rugina [7] present a novel shape analysis that first partitions the heap using region inference and then tracks updates on representative heap cells independently. While their abstraction cannot track certain global properties like the aforementioned shape analyses, they make this trade-off to obtain a very scalable shape analysis that can handle singly-linked lists. Recently, Cherem and Rugina [4] have extended this analysis to handle doubly-linked lists by including the tracking of neighbor cells. McPeak and Necula [12] identify a class of axioms that can describe many common data structure invariants and give a complete decision procedure for this class. Their technique is based on verification-condition generation and thus requires loop invariant annotations. PALE [13] is a similar system also based on verification-condition generation but instead uses monadic second-order logic. Weis *et al.* [19] have extended PALE with non-deterministic field constraints (and some loop invariant inference), which enables some reasoning of skip list structures.

Perry *et al.* [15] have also observed inductive definitions in a substructural logic could be an effective specification mechanism. They describe shape invariants for dynamic analysis with linear logic (in the form of logic programs).

## 7 Conclusion

We have described a lightweight shape analysis based on user-supplied structural invariant checkers. These checkers, in essence, provide the analysis with user-specified memory abstractions. Because checkers are only unfolded when the regions they summarize are manipulated, these specifications allow the user to focus the efforts of the analysis by enabling it to expose disjunctive memory states only when needed. The key mechanisms we utilize to develop such a shape analysis is a generalization of checker-based summaries with partial checker runs and a folding strategy based on guidance from previous iterates. In this paper, we have focused on using structural checkers to analyze algorithms that traverse the structures unidirectionally. We believe such ideas could be applicable more broadly (both in terms of utilizable checkers and algorithms analyzed).

**Acknowledgments.** We would like to thank Hongseok Yang, Bill McCloskey, Gilad Arnold, Matt Harren, and the anonymous referees for providing helpful comments on drafts of this paper.

## References

1. Arnold, G.: Specialized 3-valued logic shape analysis using structure-based refinement and loose embedding. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, Springer, Heidelberg (2006)
2. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, Springer, Heidelberg (2007)
3. Chang, B.-Y.E., Rival, X., Necula, G.C.: Shape analysis with structural invariant checkers. Technical Report UCB/EECS-2007-80, UC Berkeley (2007)
4. Cherem, S., Rugina, R.: Maintaining doubly-linked list invariants in shape analysis with local reasoning. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, Springer, Heidelberg (2007)
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
6. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, Springer, Heidelberg (2006)
7. Hackett, B., Rugina, R.: Region-based shape analysis with tracked locations. In: POPL (2005)
8. Lee, O., Yang, H., Yi, K.: Automatic verification of pointer programs using grammar-based shape analysis. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, Springer, Heidelberg (2005)
9. Magill, S., Nanevski, A., Clarke, E., Lee, P.: Inferring invariants in separation logic for imperative list-processing programs. In: Semantics, Program Analysis, and Computing Environments for Memory Management (2006)
10. Manevich, R., Sagiv, S., Ramalingam, G., Field, J.: Partially disjunctive heap abstraction. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, Springer, Heidelberg (2004)
11. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, Springer, Heidelberg (2005)
12. McPeak, S., Necula, G.C.: Data structure specifications via local equality axioms. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, Springer, Heidelberg (2005)
13. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: PLDI (2001)
14. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002 and ETAPS 2002. LNCS, vol. 2304, Springer, Heidelberg (2002)

15. Perry, F., Jia, L., Walker, D.: Expressing heap-shape contracts in linear logic. In: Generative Programming and Component Engineering (2006)
16. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. Commun. ACM 33(6) (1990)
17. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)
18. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3) (2002)
19. Wies, T., Kuncak, V., Lam, P., Podelski, A., Rinard, M.C.: Field constraint analysis. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, Springer, Heidelberg (2005)

# Footprint Analysis: A Shape Analysis That Discovers Preconditions

Cristiano Calcagno[1], Dino Distefano[2], Peter W. O'Hearn[2],
and Hongseok Yang[2]

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

**Abstract.** Existing shape analysis algorithms infer descriptions of data structures at program points, starting from a given precondition. We describe an analysis that does not require any preconditions. It works by attempting to infer a description of only the cells that might be accessed, following the footprint idea in separation logic. The analysis allows us to establish a true Hoare triple for a piece of code, independently of the context in which it occurs and without a whole-program analysis. We present experimental results for a range of typical list-processing algorithms, as well as for code fragments from a Windows device driver.

## 1 Introduction

Existing shape analysis engines (e.g., [25,9,15,13,4]) require a precondition to be supplied in order to run. Simply put, this means that they cannot be used automatically without either knowing the execution context (which might be an entire operating system, or even be unknown) or by manually supplying a precondition (which for complex code can be hard to determine). If, though, we could discover preconditions then, combined with a usual forwards-running shape analysis, we could automatically generate true Hoare triples for pieces of code independently of their context.

This paper defines *footprint analysis*, a shape analysis that is able to discover preconditions (as well as postconditions). Our results build on the work on shape analysis with separation logic [9]; the footprint analysis algorithm is itself parameterized by a standard shape analysis based on separation logic. In essence, we are leveraging the "footprint" idea of [21]. Separation logic gives us mechanisms whereby a specification can concentrate on only the cells accessed by a program, while allowing the specification to be used in wider contexts via the "frame rule". For program analysis this suggests, when considering a code fragment in isolation, to try to discover assertions that describe the footprint, rather than the entire global state of the system. This is the key idea that makes our analysis viable: the entire global state can be enormous, or even unknown, where we can use much smaller assertions to talk about the footprint.

Footprint analysis runs forwards, updating the current heap when it can in the usual way of shape analysis. However, when a dereference to a potentially-dangling

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 402–418, 2007.

pointer is encountered, that pointer is added into the “footprint assertion”, which describes the cells needed for the program to run safely. If we start the analysis with the empty heap as the initial footprint assertion then, ideally, it would find the collection of safe states, ones that do not lead to a dereference of a dangling pointer or other memory fault.

We say ideally here because there is a complication. In order to stop the footprint assertion from growing forever it is periodically abstracted. The abstraction we use is an overapproximation and, usually in shape analysis, this leads to incompleteness while maintaining soundness. But, abstracting the footprint assertion is tantamount to *weakening a precondition*, and so for us is a potentially unsound step. As a result, we also use a post-analysis phase, where we run a standard forwards shape analysis to filter out the unsafe preconditions that have been discovered. For each of the safe preconditions, we also generate a corresponding postcondition.

The source of this complication is, though, also a boon. In shape domains it can be the case that a reasonably general assertion can be obtained from a specific concrete heap using the domain’s abstraction function. For example, a linked list of length three is often abstracted as a linked list of unknown length. This nature of the shape domains is what lets footprint analysis often find a reasonably general precondition, which is synthesized from concrete assertions generated when we encounter potential memory faults.

We show by experimental results that footprint analysis is indeed able to discover non-trivial preconditions, in a number of cases resembling the precondition that we would normally write by hand. Intuitively, the algorithm works well because pointer programs are often insensitive to the abstractions we use, and so the step for filtering out unsound preconditions often does nothing. A limitation of the paper is that we do not have a thorough theoretical explanation to back this intuition up,[1] so the method might be regarded as having a heuristic character. We felt it reasonable to describe our discovery algorithm now because the results of the analysis are encouraging, and the algorithm itself employs the footprint idea in a novel way. Also, there are several potential further applications of having in place an analysis that discovers preconditions, which we describe at the end of the paper. We hope that further theoretical understanding of our method will be forthcoming in the future.

*Context and Further Discussion.* For precondition discovery one of the first things that come to mind is to use an underapproximating backwards analysis. While possible in principle, we have found it difficult to obtain precise and efficient backwards analyses for shape domains. As far as we are aware the problem of finding a useful backwards-running shape analysis is open.

Footprint analysis can be seen as an instance of the general idea of relational program analysis [7]. The purpose of a relational analysis is to compute an

[1] Mooly Sagiv has suggested starting from a $\top$ value and homing in on the needed states using a greatest fixed-point computation, as in [27]. We have not been able to make that approach work, and it does not describe what our analysis is doing, but it and other approaches are worth exploring.

overapproximation of the transition relation of a program. After the post-analysis check to filter out unsound preconditions, footprint analysis returns a set of true Hoare triples for a program, and from this it is easy to construct the relational overapproximation.

The shape analysis of [14] tracks relationships between input and output heaps. In the examples there, a precondition was typically supplied as input; for example, for in-place list reversal the input indicated an acyclic linked list. However, it might be possible to use a similar sort of idea to replace the separate preconditions and postconditions used in our algorithm, which might result in an improved precondition discovery method.

## 2 Basic Ideas

In this section we illustrate how our algorithm finds a precondition via a fixed-point calculation, using a simple example. The paper continues in the next section with the formal development.

The abstract states in the analysis consist of two assertions $(H, F)$, represented as separation logic formulae (see [22] for the basics of separation logic). $H$ represents the currently known or allocated heap and $F$ the cells that are needed (the footprint). As described above, the analysis runs forwards, adding pointers into the footprint assertion $F$ when dereferences to potentially dangling pointers are encountered. In doing this care is needed in the treatment of variables, especially what we call *footprint variables*.

The algorithm is attempting to discover a precondition that describes "safe heaps", ones that do not lead to a dereference of a dangling pointer or other memory fault when the program in question is run. We illustrate with a program that disposes all the elements in an acyclic linked list. Footprint analysis discovers the precondition pictured in Figure 1, which says that c points to a linked list segment terminating at 0. This precondition describes just what is needed in order for the program not to dereference a dangling pointer during execution. We now outline how footprint analysis finds this assertion.

We begin symbolic execution with c==c_ $\wedge$ emp as the current heap and emp as the footprint. Note that c==c_ allows for a state where c (or any other location) is dangling. The current heap includes a footprint variable c_, and assertion emp

```
while (c!=0) {
   t=c;
   c=c->tl;
   free(t);
}
```

Discovered Precondition: c==c_ $\wedge$ lseg(c_,0)

**Fig. 1.** Program delete_list, and discovered precondition when run in start state (c==c_ $\wedge$ emp, emp)

| | Current Heap | Footprint Heap |
|---|---|---|
| First iteration | | |
| **pre:** | c!=0 ∧ c==c_ ∧ t==c_ ∧ emp | emp |
| **post:** | t!=0 ∧ c==c1_ ∧ t==c_ ∧ c_ ↦ c1_ | c_ ↦ c1_ |
| Second Iteration | | |
| **pre:** | c!=0 ∧ c==c1_ ∧ t==c1_ ∧ emp | c_ ↦ c1_ |
| **post:** | t!=0 ∧ c==c2_ ∧ t==c1_ ∧ c1_ ↦ c2_ | c_ ↦ c1_ * c1_ ↦ c2_ |
| **abs post:** | t!=0 ∧ c==c2_ ∧ t==c1_ ∧ c1_ ↦ c2_ | lseg(c_,c2_) |
| Third Iteration | | |
| **pre:** | c!=0 ∧ c==c2_ ∧ t==c2_ ∧ emp | lseg(c_,c2_) |
| **post:** | t!=0 ∧ c==c3_ ∧ t==c2_ ∧ c2_ ↦ c3_ | lseg(c_,c2_) * c2_ ↦ c3_ |
| **abs post:** | t!=0 ∧ c==c3_ ∧ t==c2_ ∧ c2_ ↦ c3_ | lseg(c_,c3_) |

**Fig. 2.** Pre and Post States at line 3 during footprint analysis of delete_list

represents the empty heap. When execution enters the loop and gets to line 3, we will attempt a heap dereference to c->tl but where we do not know that c is allocated in the precondition. This is represented in the precondition for the first iteration in Figure 2. At this point the knowledge that c points to something is added to the footprint: we need that information in order for our program not to commit a memory fault. Also, though, in order to continue symbolic execution from this point, this knowledge is added to the allocated heap as well, as pictured in the postcondition for the first iteration in Figure 2. Notice that we express that c points to something in terms of the footprint variable c_. Because it is not a program variable, and so not changed by the program, this will enable us to percolate the footprint information back to the precondition.[2]

Now, the next statement in the program, line 4, removes the assertion c_ ↦ c1_ from the current heap, using the knowledge that t=c_, but that assertion is left in the footprint. Then, when we execute the second iteration of the loop we again encounter a state where c is not allocated in the current heap. At this point we again add a pointer to the footprint and to the current heap: see the pre and post for the second iteration in Figure 2. The assertion in the footprint part uses the separating conjunction *, which requires that the conjuncts hold for separate parts of memory (and so here, denote distinct cells). Notice that the footprint variable c1_ was known to equal c in the precondition. Also, in the postcondition we generate another footprint variable, c2_.

So, after two iterations, we have found a linked list of length two in the footprint. But, this way of generating new footprint variables is a potential source of divergence in the analysis. In order to enable the fixed-point calculation to converge we abstract the footprint part of the assertion, as indicated in Figure 2, and the footprint now says that there is a list segment from c_ to c2_. This

[2] Notice also, though, that an additional footprint variable c1_ is added: the footprint variables resemble those typically used for seeding initial program states, but seeding does not cover all of their uses.

abstraction step has lost the information that the list is of length two, in that the assertion is satisfied by lists of length three, four, and so on.[3]

Continuing our narrative symbolic execution, the `free` statement will delete the assertion "`c1_` $\mapsto$ `c2_`" from the current heap (but not the footprint), and when we go into the third iteration we will again try to dereference `c->tl` when it is not known to be allocated from the current heap in the `free` command. We put a $\mapsto$ assertion in the current and footprint parts again, and then abstract. Now, when we apply abstraction the assertion "`c2_` $\mapsto$ `c3_`" is swallowed into the list segment. Except for the names of newly-generated footprint variables, the abstracted post we obtained in the second iteration is the same as in the third, and we view the newly-generated footprint variables as alpha-renameable. The reader can see the relevance to fixed-point convergence.

Finally, we can exit the loop by removing "`c2_` $\mapsto$ `c3_`" from the current heap in the `free` command, and adding the negation of the loop conditional to the heap and footprint, and forgetting about `t` because it is a local variable. A bit of logic tells us that the footprint part is equivalent to `c3_ == 0` $\wedge$ `lseg(c_,c3_)`, and when we add this to the initial precondition `c==c_` we obtain the overall precondition pictured in Figure 1.

## 3 Programming Language and Generic Analysis

In this section we define the programming language used in the formal part of the paper. We also set up a generic analysis, following the tradition of abstract interpretation [8], which will have the shape and footprint analyses described later as instances.

*Programming Language.* In the paper, we use a simple `while` language extended with heap operations:

$$\begin{array}{lll} E, F & ::= & x \mid f(E_1, \ldots, E_n) \\ b & ::= & E = F \mid E \neq F \\ a[E] & ::= & [E] := F \mid \mathsf{dispose}(E) \mid x := [E] \\ a & ::= & x := E \mid x := \mathsf{new}(E) \\ c & ::= & a[E] \mid a \mid c_1; c_2 \mid \mathsf{if}\ b\ c_1\ c_2 \mid \mathsf{while}\ b\ c \end{array}$$

An expression $E$ is either a variable or a heap-independent term $f(E_1, \ldots, E_n)$. The language has two classes of atomic commands. $a[E]$ attempts to dereference cell $E$, updating it ($[E] := F$), disposing it ($\mathsf{dispose}(E)$), or reading its content ($x := [E]$). The other atomic commands, denoted $a$, do not access existing cells.

[3] This step of abstraction depends on which abstract domain we plug into our footprint analysis; several have appeared in the literature, and the footprint analysis does not depend on any one choice. In this example, we have assumed that the "lseg" predicate describes "possibly circular list segments", which allows the abstraction step we have done. If circularity were outlawed in our abstract domain, as in the particular domain of [9], then one more loop iteration would be needed before abstraction could occur.

*Generic Analysis.* The analyses in this paper will use the topped powerset $\mathcal{P}^\top(S)$ of a set $S$; i.e., the powerset with an additional greatest element. A set $X \in \mathcal{P}(S)$ represents a disjunction of its elements $x \in X$, and $\top$ indicates that the analysis detected an error in a given program. When $D = \mathcal{P}^\top(S)$, we call $S$ the underlying set of the abstract domain $D$.

Given function $t\colon S \to \mathcal{P}^\top(S')$ and partial or total function $f\colon S \rightharpoonup S'$, we can lift them to functions $t^\dagger, \mathcal{P}^\top(f) : \mathcal{P}^\top(S) \to \mathcal{P}^\top(S')$ by

$$\begin{aligned} t^\dagger(X) &\stackrel{\text{def}}{=} \textbf{if } (X = \top) \textbf{ then } \top \textbf{ else } \textstyle\bigsqcup_{x \in X} t(x) \\ \mathcal{P}^\top(f)(X) &\stackrel{\text{def}}{=} \textbf{if } (X = \top) \textbf{ then } \top \textbf{ else } \{f(x) \mid x \in X\}. \end{aligned}$$

The generic analysis framework consists of the following data.

(1) A set $S$ of abstract states, inducing the abstract domain $D = \mathcal{P}^\top(S)$, which forms a complete lattice $(D, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$.

(2) For all boolean expressions $b$, atomic commands $a, a[E]$ and expressions $E$, the operators

$$\mathsf{rearr}(E)\colon S \to \mathcal{P}^\top(S[E]), \quad \mathsf{filter}(b)\colon S \rightharpoonup S,$$
$$\mathsf{exec}(a[E])\colon S[E] \to S, \quad \mathsf{exec}(a)\colon S \to S, \quad \mathsf{abs}\colon S \to S.$$

Here $S[E]$ is a subset of $S$ (for all $E$), and it consists of abstract states where cell $E$ is explicitly represented by a points-to fact $E{\mapsto}E'$.

This framework does not ask for transfer functions to be given directly, but rather asks for more refined ingredients, out of which transfer functions are usually made in shape analysis. $\mathsf{rearr}(E)$ typically takes an abstract state and attempts to "concretize" cell $E$, making it a points-to fact of the form $E{\mapsto}E'$. When instantiating the generic analysis with the one in [9], this operation corresponds to unwinding an inductive definition, and when instantiating with [24] it is the materialization of a summary node. The abstraction map $\mathsf{abs}$ simplifies states, as illustrated in the example in the previous section. In [9,25] it is called canonicalization. $\mathsf{filter}(b)$ is used to filter states that do not satisfy boolean condition $b$, and $\mathsf{exec}(a[E])$ and $\mathsf{exec}(a)$ implement update (after rearrangement).

Given this data, abstract transfer functions of the primitive commands are:[4]

$$[\![b]\!] \stackrel{\text{def}}{=} \mathcal{P}^\top(\mathsf{filter}(b))$$
$$[\![a[E]]\!] \stackrel{\text{def}}{=} (\mathcal{P}^\top(\mathsf{abs} \circ \mathsf{exec}(a[E])) \circ \mathsf{rearr}(E))^\dagger \qquad [\![a]\!] \stackrel{\text{def}}{=} \mathcal{P}^\top(\mathsf{abs} \circ \mathsf{exec}(a)).$$

The execution of a command accessing $E$ is done in three steps: first the cell $E$ is exposed by $\mathsf{rearr}(E)$, then the state is updated according to the semantics of the command $a[E]$ by $\mathsf{exec}(a[E])$ and finally the resulting state is abstracted by $\mathsf{abs}$. The execution of a command $a$ that does not access the heap does not

[4] Our analysis specification presumes that abstraction is applied after every transfer function, but it is also possible to instead take it out of the transfer functions and apply only often enough to allow the loop computations to converge.

involve the rearrangement phase. The reader is referred to [9] for an extensive treatment of transfer functions defined in terms of $\mathsf{rearr}$, $\mathsf{exec}$ and $\mathsf{abs}$.

We may then define monotone functions $[\![c]\!]\colon D \to D$ for each command $c$ in the usual way of abstract interpretation.[5]

$$[\![c_1; c_2]\!] = [\![c_2]\!] \circ [\![c_1]\!] \qquad [\![\mathsf{if}\ b\ c_1\ c_2]\!](d) = ([\![c_1]\!] \circ [\![b]\!])(d) \sqcup ([\![c_2]\!] \circ [\![\neg b]\!])(d)$$
$$[\![\mathsf{while}\ b\ c]\!](d) = [\![\neg b]\!](\mathsf{lfix}\ \lambda d'.\ d \sqcup ([\![c]\!] \circ [\![b]\!])(d'))$$

## 4 Underlying Shape Analysis Based on Separation Logic

We assume that we are given three disjoint countable sets of variables:

- $\mathsf{Vars}$ for program variables $x, y$;
- $\mathsf{PVars}$ for primed variables $x', y'$;
- $\mathsf{FVars}$ for footprint variables $\overline{x}, \overline{y}$.

In the following, primed variables will be used as notation for existential quantified variables whereas footprint variables will be used to store initial values of program variables or cells.

Let $\mathsf{Locs}$ and $\mathsf{Vals}$ be countable infinite sets of locations and values, respectively, such that $\mathsf{Locs} \subseteq \mathsf{Vals}$. When $V$ is set to be the union of $\mathsf{Vars}$, $\mathsf{PVars}$ and $\mathsf{FVars}$, our concrete storage model is given by:

$$\mathsf{Stacks} \stackrel{\mathrm{def}}{=} V \to \mathsf{Vals} \quad \mathsf{Heaps} \stackrel{\mathrm{def}}{=} \mathsf{Locs} \rightharpoonup_{\mathsf{fin}} \mathsf{Vals} \quad \mathsf{States} \stackrel{\mathrm{def}}{=} \mathsf{Stacks} \times \mathsf{Heaps}.$$

Each state consists of stack and heap components. The stack component $s$ records the values of program, primed and footprint variables, and the heap component $h$ specifies the identities and contents of allocated cells. Note that this model can allow data structures of complex shape, because a pair of addresses can be a value so a cell can have two outgoing pointers.

The analysis described in this paper uses separation logic assertions (called *symbolic heaps*) to represent abstract states. Symbolic heaps $H$ are given by the following grammar:

$$\begin{aligned}
E, F &::= \mathsf{nil} \mid x \mid x' \mid \bar{x} \mid \cdots \\
\Pi &::= \mathsf{true} \mid E = E \mid E \neq E \mid \Pi \wedge \Pi \mid \cdots \\
\Sigma &::= \mathsf{true} \mid \mathsf{emp} \mid E \mapsto E \mid \Sigma * \Sigma \mid \cdots \\
H &:= \Pi \wedge \Sigma
\end{aligned}$$

Intuitively, in a symbolic heap $\Pi \wedge \Sigma$, the first conjunct $\Pi$ contains only expressions describing the relations among program, primed and footprint variables given by the stack whereas $\Sigma$ describes the allocated heap. The predicate $E \mapsto F$ is true when the cell $E$ is allocated, its value is $F$, and nothing else is allocated.

[5] Our requirement of a complete lattice and monotonicity can be weakened if we include a widening operator.

$\Sigma_1 * \Sigma_2$ holds when the heap can be split into components, one of which makes $\Sigma_1$ true and the other of which makes $\Sigma_2$ true. See [22]. We assume that primed variables in each symbolic heap $H$ are existentially quantified.

The use of $\cdots$ is to allow for various other predicates, such as for list segments and for trees. In this sense, the present section is setting down a parameterized analysis which can be instantiated, e.g., by [3,5,16]. More importantly, we are emphasizing that our footprint analysis algorithm (in the next section) is not tied to any of these particular analyses.

We define a "separation-logic-based shape analysis" to consist of the following.

1. An instance $(S, \{S[E]\}_E, \mathsf{rearr}, \mathsf{filter}, \mathsf{exec}, \mathsf{abs})$ of the generic analysis from Section 3.
2. The shape analysis should use separation logic, in the style of [9]. This means that the underlying set $S$ of the abstract domain consists of sets of symbolic heaps, and that for each expression $E$, all the symbolic heaps in the subset $S[E]$ of $S$ are of the form $\Pi \wedge (E \mapsto F) * \Sigma$. We say that cell $E$ is exposed by the points-to relation.
3. A sound theorem prover $\vdash$ for proving entailments between symbolic heaps.
4. For each symbolic heap $\Pi \wedge \Sigma$ in $S$ and fresh footprint variable $\overline{x}$, the new symbolic heap $\Pi \wedge (E \mapsto \overline{x}) * \Sigma$ is in $S[E]$, or it can be shown to be inconsistent by the given theorem prover.
5. None of $\mathsf{rearr}$, $\mathsf{filter}$, $\mathsf{exec}$ and $\mathsf{abs}$ introduces new footprint variables into given symbolic heaps.
6. Writing $G$ for the set of symbolic heaps in $S$ containing only footprint variables, $\mathsf{abs}$ maps elements of $G$ to $G$. Moreover, for all $\Pi_0$ with footprint variables only, if $\Pi \wedge \Sigma$ is in $G$, then $\Pi_0 \wedge \Pi \wedge \Sigma$ is in $G$, unless it is proved to be inconsistent by the theorem prover.

## 5 Footprint Analysis

Now suppose we are given a separation-logic-based shape analysis as specified in the last section. Recall that $S$ is the set of symbolic heaps and $G$ is the set of symbolic heaps whose only free variables are footprint variables. Our footprint analysis is an instance of the generic analysis in Section 3, where the abstract domain of our algorithm is the topped powerset

$$\mathcal{P}^{\top}(S \times G).$$

A pair $(H, F)$ in $S \times G$ represents the current heap $H$ and the computed footprint $F$ at the current program point. Note that the footprint can contain footprint variables only. The algorithm relies on this requirement to ensure that the computed footprint is a property of the initial states, rather than the states at the current program point.

We specify our algorithm by defining the data required by the generic analysis, which we call $\mathsf{newRearr}, \mathsf{newFilter}, \mathsf{newExec}, \mathsf{newAbs}$ in order to avoid confusion with the abstract transfer functions of the given underlying shape analysis, which the footprint analysis will be defined in terms of.

First, we give the definition of newRearr, in terms of the rearrangement rearr of the given shape analysis:

```
$\mathsf{newRearr}(E) \;:\; S \times G \to \mathcal{P}^{\top}(S[E] \times G)$
$\mathsf{newRearr}(E)(H, F) \overset{\text{def}}{=}$ let $\mathcal{H} = \mathsf{rearr}(E)(H)$
    in if $\neg\,(\mathcal{H}{=}\top)$ then $\{(H', F) \mid H' \in \mathcal{H}\}$
      else if ($H \vdash E{=}\overline{x}_0$ for some footprint var $\overline{x}_0$) and
              $\neg(F * \overline{x}_0{\mapsto}\overline{x}_1 \vdash \mathsf{false}$ for some fresh $\overline{x}_1)$
        then $\{(H * E{\mapsto}\overline{x}_1,\; F * \overline{x}_0{\mapsto}\overline{x}_1)\}$
        else $\top$
```

This subroutine takes two symbolic heaps, $H$ for the overapproximation of the reachable states and $F$ for the footprint, and exposes a specified cell $E$ from $H$. Intuitively, it first calls the rearrangement step of the underlying shape analysis to prove that a dereferenced cell $E$ is allocated. In case this first attempt fails, the subroutine adds the missing cell to the footprint and the current symbolic heap. This is the point at which the underlying shape analysis would have stopped, reporting a fault. Note that before adding the points-to relation to $F$, the subroutine checks whether $E$ can be rewritten in terms of a footprint variable $\overline{x}_0$. This ensures that the computed footprint is independent of the values of variables whose value changes (program variables) or is determined during execution (primed variables).

Next, we define the subroutine newFilter:

```
$\mathsf{newFilter}(b) \;:\; S \times G \rightharpoonup S \times G$
$\mathsf{newFilter}(b)(H, F) \overset{\text{def}}{=}$ if ($\mathsf{filter}(b)(H)$ is not defined) then undefined
    else let $H' = \mathsf{filter}(b)(H)$
      in if $\neg\,(H \vdash b{\Leftrightarrow}\overline{b}$ for some $\overline{b}$ with footprint vars only)
        then $(H', F)$
        else $(H', \overline{b} \wedge F)$
```

This subroutine tries to rewrite $b$ in terms of footprint variables only. If it succeeds, the rewriting gives an additional precondition $\overline{b}$ that will make the test $b$ hold: the computation can then pass through the filter, and the result of the rewriting is conjoined to the footprint. On the other hand, if the rewriting fails, the analyzer keeps the given footprint $F$.

Finally, the subroutine newExec is defined by the execution of exec for the first component $H$ for shape invariants.

$$\mathsf{newExec}(a[E])(H, F) \overset{\text{def}}{=} (\mathsf{exec}(a[E])(H), F) \quad \mathsf{newExec}(a)(H, F) \overset{\text{def}}{=} (\mathsf{exec}(a)(H), F)$$

And newAbs is defined by applying abstraction to both the shape and footprint:

$$\mathsf{newAbs}(H, F) \overset{\text{def}}{=} (\mathsf{abs}(H), \mathsf{abs}(F))$$

### 5.1 Hoare Triple Generation

We show how the footprint analysis algorithm can be used to generate true Hoare triples. First there is a pre-processing step which generates an initial symbolic heap that saves the initial values of program variables into footprint variables. Then, after running footprint analysis, we run a post-processing step which takes the output of our algorithm and, for each computed precondition, it runs the underlying shape analysis to compute the appropriate postcondition.

Let $x_1, \ldots, x_n$ be program variables that appear in a given program $c$. Write $[\![-]\!]_s$ for the given shape analysis and $[\![-]\!]_f$ for the corresponding footprint analysis. Formally, the Hoare triple generation for a program $c$ works as follows:

```
let $\Pi_0 \overset{\mathrm{def}}{=} (x_1{=}\overline{x}_1 \wedge \ldots \wedge x_n{=}\overline{x}_n)$ and $\mathcal{F} \overset{\mathrm{def}}{=} [\![c]\!]_f(\{(\Pi_0 \wedge \mathsf{emp},\ \mathsf{emp})\})$
in if $(\mathcal{F}{=}\top)$ then report the possibility of a catastrophic fault
  else
    $\big\{\{F'\}c\{\bigvee_{H' \in \mathcal{H}} H'\} \mid (H,F) \in \mathcal{F} \wedge F'{=}\mathsf{ren}(\Pi_0 \wedge F) \wedge \mathcal{H}{=}[\![c]\!]_s(\{F'\}) \wedge \mathcal{H}{\neq}\top\big\}$.
```

Here $\mathsf{ren}(\Pi_0 \wedge F)$ renames all the footprint variables by primed variables.

If the underlying shape analysis is sound with respect to a concrete semantics of a programming language then we automatically get true Hoare triples. However, it would be easy to generate *some* true Hoare triples, if we were content to generate precondition `false`. What our algorithm is aiming at is to generate preconditions that cover as many "safe states" as possible, ones which ensure that the program will not commit a memory fault. There can be, of course, no perfect such algorithm for computability reasons. In our case, though, it is well to mention two possible sources of inaccuracy.

First, because the analysis applies abstraction to the footprint (the eventual precondition), this can lead us outside of the safe states (it is essentially *weakening* a precondition). We have found that it very often leads to safe preconditions in our experimental results. An intuitive reason for this is that the safety of typical list programs is often insensitive to the abstraction present in shape analyses. But, because this "often" is not "always", as we will see in the next section, the Hoare triple generation just described filters out these unsafe pre-states by calling the (assumed to be sound) underlying shape analysis.

Second, our algorithm does not perform as much case analysis on the structure of heap as is theoretically possible, and this leads to incompleteness (where fewer safe states are described than might otherwise be). We have made this choice for efficiency reasons. We believe that our experimental results in the next section show that this is not an unrealistic engineering decision. But we also discuss an example (`append.c`) where the resulting incompleteness arises.

Finally, we point out that from true Hoare triples computed by our analysis, one can easily construct a relational overapproximation of the transition relation of a program. Suppose that our analysis generated a set $\big\{\{P_i\}c\{Q_i\}\big\}_{i \in I}$ of true Hoare triples for a given program $c$. Then, by [6], there is a state transformer $r$ (i.e., relation from $\mathsf{States}$ to $\mathsf{States} \cup \{\mathsf{wrong}\}$) with the following three properties: (1) the transformer $r$ satisfies triple $\{P_i\}r\{Q_i\}$ for all $i \in I$; (2) it satisfies the locality conditions in separation logic[6]; and (3) the transformer overapproximates all the other state transformers satisfying (1) and (2) (i.e., it is bigger

[6] The locality conditions are safety monotonicity and frame property in [28].

than those state transformers according to the subset ordering.) Indeed, [6] gives an explicit definition of the transformer $r$.[7] This transformer overapproximates the relational meaning of program $c$, since all the triples $\{P_i\}c\{Q_i\}$ hold for $c$ and the meaning of $c$ satisfies the locality conditions.

## 6 Experimental Results

Our experimental results are for an implementation of our analysis developed using the CIL infrastructure [19]. We used two abstract domains for the experiments, one based on the simple list domain in [9] and the other with the domain of [2] which uses a higher-order variant of the list segment predicate to describe composite structures.

*List program examples.* Table 1 shows the results of applying the footprint analysis to a set of list programs taken from the literature.[8] The Disjuncts column reports the number of disjuncts of the computed preconditions. Amongst all the computed preconditions, some can be unsafe and there can be redundancy in that one can imply another. The Unsafe Pre column indicates the preconditions filtered out when we re-execute the analysis. In the Discovered Precondition column we have dropped the redundant cases and used implication to obtain a compact representation that could be displayed in the table. For the same precondition, the table shows different disjuncts on different lines. For all tests except one (`merge.c`, discussed below) our analysis produced a precondition from which the program can run safely, without generating a memory fault, obtaining a true Hoare triple. We comment on a few representative examples.

`del-doublestar` uses the usual C trick of double indirection to avoid unnecessary checking for the first element, when deleting an element from a list.

```
void del-doublestar(nodeT **listP, elementT value)
{  nodeT *currP, *prevP;
  prevP=0;
  for (currP=*listP; currP!=0; prevP=currP, currP=currP->next) {
    if (currP->elmt==value) {  /* Found it. */
      if (prevP==0) *listP=currP->next;
      else prevP->next=currP->next;
      free(currP);
} } }
```

[7] Formally, $r \subseteq \mathsf{States} \times (\mathsf{States} \cup \{\mathsf{wrong}\})$ is defined by:

$$\begin{aligned} (s,h)[r]\mathsf{wrong} &\iff \forall i \in I.\, (s,h) \notin [\![P_i * \mathsf{true}]\!] \\ (s,h)[r](s',h') &\iff \forall i \in I.\, \forall h_0, h_1.\, (s,h_0) \in [\![P_i]\!] \wedge h_0 \bullet h_1 = h \\ &\qquad\qquad \implies \exists h_0'.\, (s',h_0') \in [\![Q_i]\!] \wedge h_0' \bullet h_1 = h'. \end{aligned}$$

where $[\![P_i]\!]$, $[\![Q_i]\!]$ are the usual meaning of assertions and $\bullet$ is a partial heap-combining operator in separation logic.

[8] In some cases the reported memory consumption was exactly the same for different programs; this happens because the memory chunks allocated by OCAML's runtime system are too coarse to observe small differences between example programs.

The first disjunct of the discovered precondition is

```
listP|->x_ * ls(x_,x1_) * x1_|->elmt:value
```

This shows the cells that are accessed when the element being searched for happens to be in the list. Note that it does not record list items which might follow the value: they are not accessed.[9] A postcondition for this precondition has just a list segment running to x2_:

```
listP|->x_ * ls(x_,x2_)
```

The other precondition

```
listP|->x_ * ls(x_,0)
```

corresponds to when the element being searched for is not in the list. The algorithm fails to discover a circular list in the precondition

```
listP|->x_ * ls(x_,x_)
```

The program infinitely loops on this input, but does not commit a memory safety violation. This is an example of incompleteness in our algorithm.[10]

Further issues can be seen by contrasting append.c and append-dispose.c. The former is the typical algorithm for appending two lists x and y. The computed precondition is

```
ls(x_,0)
```

Again, notice that nothing reachable from y is included, as the appended list is not traversed by the algorithm: it just swings a pointer from the end of the first list. However, when we post-compose appending with code to delete all elements in the acyclic list rooted at x, which is what append-dispose.c does, then the footprint requires an acyclic list from y as well

```
ls(x,0) * ls(y,0)
```

The only program for which we failed to find a safe precondition was merge.c, the usual program to merge two sorted lists: instead, footprint analysis returned all unsafe disjuncts (which were pruned at re-execution time). The reason is that our analysis essentially assumes that the safety of the program is insensitive to the abstraction performed in the analysis, and this is false for merge.c.

[9] This point could be relevant to interprocedural analysis, where [23,10] pass a useful but coarse overapproximation of the footprint to a procedure, consisting of all abstract nodes reachable from certain roots.

[10] Note that the problem here does not have to do with circular lists *per se*, as our algorithm succeeds in finding preconditions for algorithms for circular linked lists (e.g., traverse-circ.c); rather, it has to do with incompleteness arising from avoidance of case analysis mentioned in Section 5.1.

**Table 1.** Experimental results for list programs

| Program | Time (sec) | Space (Mb) | # of Disj. | Uns Pre | Discovered Precondition |
|---|---|---|---|---|---|
| sappend.c | 0.035 | 0.74 | 4 | 0 | ls(x,0) |
| append-dispose.c | 0.099 | 0.74 | 17 | 0 | ls(x,0)*ls(y,0) |
| copy.c | 0.031 | 0.74 | 4 | 0 | ls(c,0) |
| create.c | 0.014 | 0.49 | 1 | 0 | emp |
| del-doublestar.c | 0.045 | 0.49 | 10 | 0 | listP↦x_*ls(x_,x1_) *x1_↦elmt:value,<br>listP↦x_*ls(x_,0) |
| del-all.c | 0.014 | 0.49 | 4 | 0 | ls(c,0) |
| del-all-circular.c | 0.015 | 0.49 | 3 | 0 | c↦c_*ls(c_,c) |
| del-lseg.c | 0.639 | 1.23 | 48 | 0 | z≠0∧ls(c,z)*ls(z,0),<br>z≠w∧ls(c,z)*ls(z,w)*w↦0,<br>z≠w∧w_≠0∧ls(c,z)*ls(z,w)*w↦w_,<br>z≠c∧c↦0,<br>z≠c∧z≠c_∧c↦c_*ls(c_,0),<br>c=0∧emp |
| find.c | 0.057 | 0.74 | 12 | 0 | ls(c,b)*b↦0,<br>b_≠0∧ls(c,b)*b↦b_,<br>b≠c∧b≠c_∧c↦c_*lseg(c_,0),<br>b≠c∧c↦0,<br>c=0∧emp |
| insert.c | 0.170 | 0.74 | 10 | 0 | e1≠0∧e2≠0∧c_≠d_∧<br>c↦c_*ls(c_,d_)*d↦dta:e3,<br>e1≠0∧e2≠0∧c_≠0∧c↦c_*ls(c_,0),<br>e1≠0∧c↦0,<br>e1≠0∧e2=0∧c↦c_*c_↦-,<br>e1=0∧c↦-,<br>c=0∧emp |
| merge.c | 0.561 | 1.47 | 30 | 30 | — |
| reverse.c | 0.020 | 0.74 | 4 | 0 | ls(c,0) |
| traverse-circ.c | 0.013 | 0.49 | 3 | 0 | c↦c_*ls(c_,c) |

*IEEE 1394 firewire driver routines.* We then changed the abstract domain in our implementation, swapping the simple list domain for the domain from [2]. Table 2 reports experimental results on several routines from a firewire driver for Windows.[11] We emphasize that the ability of that domain to analyze the driver code is not a contribution of the present paper: it was already shown in [2] when preconditions were generated by environment code or supplied manually. Here, we are just using that domain with our footprint analysis algorithm.

The procedure `t1394Diag_PnpRemoveDevice`, for which our analysis timed out, has five while loops, two of which are nested, and multiple nested conditionals.

[11] After dropping redundant disjunct and simplify by implication, the precondition for device drivers remain still considerably large. Therefore, due to space limitations, in this table, we do not report the discovered preconditions.

**Table 2.** Experimental results from firewire device driver routines

| Program | Time (s) | Memory | # of Disjuncts | Unsafe Pre |
|---|---|---|---|---|
| t1394Diag-CancelIrp.c | 0.08928 | 1.23Mb | 11 | 2 |
| t1394Diag-CancelIrpFix.c | 0.20461 | 1.23Mb | 10 | 0 |
| t1394Diag_PnpRemoveDevice | T/O | — | — | — |
| t1394-BusResetRoutine.c | 0.14924 | 1.23Mb | 4 | 0 |
| t1394-GetAddressData.c | 0.08692 | 1.23Mb | 9 | 2 |
| t1394-GetAddressDataFix.c | 0.08906 | 1.23Mb | 3 | 0 |
| t1394-IsochDetachCompletionRoutine.c | 1.76640 | 2.70Mb | 39 | 0 |
| t1394-SetAddressData.c | 0.06614 | 1.23Mb | 9 | 1 |
| t1394-SetAddressDataFix.c | 0.12242 | 1.23Mb | 9 | 0 |

At the time of writing, our analysis does not implement several optimizations for scalability. For example, we have not yet implemented the acceleration techniques based on widening from [5].

For five (out of nine) of these routines our analysis found only sound preconditions from which it is ensured the program will run safely. For three of these routines (`t1394Diag_CancelIrp`, `t1394_GetAddressData`, `t1394_SetAddressData`) for which it was known to have memory errors (see [2] for details), our analysis found two kinds of preconditions:

- *Safe preconditions* that exclude the errors. The analyzer generated true Hoare triples for these preconditions.
- *Unsafe preconditions* that lead to (in this case) known memory errors. For these analyzed routines the memory errors occur when they are given empty lists. All of these unsafe empty-list cases are included in the discovered preconditions. But, they are the only reasons for the preconditions to be unsafe; if we semantically rule out these empty-list cases from these preconditions by altering them manually, the preconditions cover only safe states as can be confirmed by re-execution.

The errors were fixed in `t1394Diag_CancelIrpFix`, `t1394_GetAddressDataFix` and `t1394_SetAddressDataFix`, such that the routines run safely even for the empty-list cases. The analysis correctly discovered this fact, by computing safe preconditions that include empty-list cases (in addition to all the other cases in the safe preconditions for the original routines).

## 7 Conclusion

We have presented a shape analysis that is able to discover preconditions, and we have presented initial experimental results. We are not aware of another published shape analysis that discovers preconditions (which is why we have not compared our analysis or experimental results to other work in shape analysis).

We have used two abstract domains in our experiments, one for simple linked lists and another for composite structures [9,2], but others could be used as well as long as they possess the basic separation logic structure that drives our analysis. Several other abstract domains based on separation logic formulae have been described [3,5,16,12], and we expected that other shape domains that have appeared (e.g., [25,15,13,4,18,1]) could be modified to have the requisite structure. This would not require using separation logic formulae literally, but rather needs the abstract domain to reflect the the partial commutative monoid of heap composition used in its semantics (as was done in [26,17]).

One of the basic ideas used in our analysis, that of abstracting preconditions as well as postconditions, could conceivably be replayed for other abstract domains than our shape domains. It would require more work to investigate for any given domain. We emphasize, though, that this general idea is not a significant contribution of the present paper, and we make no claims about application to other sorts of domain (e.g., numerical domains). Rather, the main contribution is the way that the footprint idea is used to design a particular family of shape analyses that discover preconditions, and the demonstration that some of the resulting analyses can possess a non-trivial degree of precision.

Footprint analysis has potential benefits for speeding up and improving the accuracy of interprocedural and concurrency shape analyses. With the footprint analysis one might analyze several procedures independently, and then use the results as (partial) summaries to avoid (certain) recomputations in an (even whole program) interprocedural analysis [23,10]. A thread-modular concurrency analysis has recently been defined [11]. The logic upon which it is based [20] requires preconditions for concurrent processes, but in [11] this issue is skirted by assigning the empty heap as a precondition to each concurrent process: the ideas here might be used to extend that analysis. We add that there are numerous technical problems to be overcome for this potential to be realized, such as the right treatment of cutpoints [23] together with footprints.

A further off possible application is when the calling context is not even available, or very large (e.g., an operating system): one might use footprint analysis to analyze code fragments that would otherwise require a whole-program analysis. Our experiments give initial indications on such an idea, but more work is needed to evaluate its ultimate viability. Conversely, there is the persistent problem of analyzing programs that themselves call other unknown or as yet unwritten procedures. It would be conceivable to use footprint analysis to treat the unknown procedures as “black holes”, where one starts footprint analysis again after a black hole to discover a precondition for the code that comes after; this would then function as a postcondition for the procedure call itself.

We do not mean to imply that the use of footprint analyses in these areas is in any way straightforward, and only hope that this work might help to spur further developments towards obtaining truly modular shape analyses.

*Acknowledgments.* We are grateful to Byron Cook, Noam Rinetzky and Mooly Sagiv for discussions. This research was supported by the EPSRC.

## References

1. Balaban, I., Pnueli, A., Zuck, L.: Shape Analysis by Predicate Abstraction. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 164–180. Springer, Heidelberg (2005)
2. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., Yang, H.: Shape analysis of composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, Springer, Heidelberg (2007)
3. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.: Automatic termination proofs for programs with shape-shifting heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 386–400. Springer, Heidelberg (2006)
4. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Tree Regular Model Checking of Complex Dynamic Data Structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006)
5. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 182–203. Springer, Heidelberg (2006)
6. Calcagno, C., O'Hearn, P., Yang, H.: Local action and abstract separation logic. In: LICS'07 (to appear, 2007)
7. Cousot, P., Cousot, R.: Modular Static Program Analysis. In: Horspool, R.N. (ed.) CC 2002 and ETAPS 2002. LNCS, vol. 2304, pp. 159–178. Springer, Heidelberg (2002)
8. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th POPL, pp. 238–252 (1977)
9. Distefano, D., O'Hearn, P., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
10. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 240–260. Springer, Heidelberg (2006)
11. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: PLDI 2007 (to appear, 2007)
12. Guo, B., Vachharajani, N., August, D.: Shape analysis with inductive recursion synthesis. In: PLDI 2007 (to appear, 2007)
13. Hackett, B., Rugina, R.: Region-based shape analysis with tracked locations. In: 32nd POPL, pp. 310–323 (2005)
14. Jeannet, B., Loginov, A., Reps, T., Sagiv, M.: A relational approach to interprocedural shape analysis. In: TR 1505, Comp. Sci. Dept., Univ. of Wisconsin (2004)
15. Lev-Ami, T., Immerman, N., Sagiv, M.: Abstraction for shape analysis with fast and precise transfomers. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 547–561. Springer, Heidelberg (2006)
16. Magill, S., Nanevski, A., Clarke, E., Lee, P.: Inferring invariants in Separation Logic for imperative list-processing programs. In: 3rd SPACE Workshop (2006)
17. Manevich, R., Berdine, J., Cook, B., Ramalingam, G., Sagiv, M.: Shape analysis by graph decomposition. In: 13th TACAS (2007)
18. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, M.: Predicate abstraction and canonical abstraction for singly-linked lists. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 181–198. Springer, Heidelberg (2005)

19. Necula, G., McPeak, S., Rahul, S., Weimer, W.: CIL:intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002 and ETAPS 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
20. O'Hearn, P.: Resources, concurrency and local reasoning. Theoretical Computer Science. Preliminary version appeared in CONCUR'04 (to appear, 2007)
21. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
22. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th LICS, pp. 55–74 (2002)
23. Rinetzky, N., Bauer, J., Reps, T., Sagiv, M., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: 32nd POPL, pp. 296–309 (2005)
24. Sagiv, M., Reps, T., Wilhelm, R.: Solving Shape-Analysis Problems in Languages with Destructive Updating. ACM TOPLAS 20(1), 1–50 (1998)
25. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM TOPLAS 24(3), 217–298 (2002)
26. Sims, É.-J.: An abstract domain for separation logic formulae. In: 1st EAAI, pp. 133–148 (2006)
27. Yahav, E., Reps, T., Sagiv, M., Wilhelm, R.: Verifying temporal heap properties specified via evolution logic. In: Degano, P. (ed.) ESOP 2003 and ETAPS 2003. LNCS, vol. 2618, pp. 204–222. Springer, Heidelberg (2003)
28. Yang, H., O'Hearn, P.: A semantic basis for local reasoning. In: Nielsen, M., Engberg, U. (eds.) ETAPS 2002 and FOSSACS 2002. LNCS, vol. 2303, pp. 402–416. Springer, Heidelberg (2002)

# Arithmetic Strengthening for Shape Analysis*

Stephen Magill[1], Josh Berdine[2], Edmund Clarke[1], and Byron Cook[2]

[1] Carnegie Mellon University
[2] Microsoft Research

**Abstract.** Shape analyses are often imprecise in their numerical reasoning, whereas numerical static analyses are often largely unaware of the shape of a program's heap. In this paper we propose a lazy method of combining a shape analysis based on separation logic with an arbitrary arithmetic analysis. When potentially spurious counterexamples are reported by our shape analysis, the method constructs a purely arithmetic program whose traces over-approximate the set of counterexample traces. It then uses this arithmetic program together with the arithmetic analysis to construct a refinement for the shape analysis. Our method is aimed at proving properties that require comprehensive reasoning about heaps together with more targeted arithmetic reasoning. Given a sufficient precondition, our technique can automatically prove memory safety of programs whose error-free operation depends on a combination of shape, size, and integer invariants. We have implemented our algorithm and tested it on a number of common list routines using a variety of arithmetic analysis tools for refinement.

## 1 Introduction

Automatic formal software verification tools are often designed either to prove arithmetic properties (*e.g. is* `x` *always greater than 0 at program location 35?*) or data structure properties (*e.g. does* `p` *always point to a well-formed list at program location 45?*). Shape analyses are developed to reason about the linked structure of data on the heap, while arithmetic analyses are designed to reason about the relationships between integer values manipulated by a program. Since integers can be stored in the heap and certain properties of data structures (such as the length of lists) are integer valued, there is non-trivial interaction between the two theories. Thus, combining a shape analysis and an arithmetic analysis is not just a matter of applying each analysis separately.

* This research was sponsored by the International Collaboration for Advancing Security Technology (iCAST), the Gigascale Systems Research Center (GSRC), the Semiconductor Research Corporation (SRC) under grant TJ-1366, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, the Defense Advanced Research Projects Agency, the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, and the General Motors Collaborative Research Lab at CMU. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution.

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 419–436, 2007.

We propose a new technique for combining a shape analysis based on *separation logic* [25] with an arbitrary arithmetic analysis. The combination technique operates by using the arithmetic analysis as a back-end for processing abstract counterexamples discovered during the shape analysis. Our shape analysis is based on those described in [4] and [22]. It is an application of abstract interpretation [11] where the abstract domain uses a fragment of separation logic. As in [4], we assume that the shape analysis supports arithmetic reasoning in its symbolic execution engine, but does not maintain enough arithmetic information in its widening step. To refine this widening step will be the job of the arithmetic analysis tool.

The shape analysis communicates with the arithmetic analysis via *counterexample programs*—integer programs that represent the arithmetic content of the abstract counterexamples. Because the language of communication consists of integer programs, any integer analysis tool can be used without modification to strengthen our shape analysis. Viewed another way, this technique allows any tool targeting integer programs to be applied—again without modification—to programs that manipulate the kinds of heap-based data structures that our shape analysis supports.

In summary, we present a new combination of shape and arithmetic analyses with the following novel collection of characteristics:

– Any arithmetic analysis can be used. The combination is not tied to any particular verification paradigm, and we can use tools based on abstract interpretation, such as ASTRÉE[7], just as easily as those based on model checking, such as BLAST[19], SLAM[2], and ARMC[24].
– The arithmetic analysis explicitly tracks integer values which appear quantified in the symbolic states but are absent in the concrete states, such as list lengths. This use of new variables in the arithmetic program to reason about quantified values makes soundness of the combination technique non-obvious. This conjunction under quantifiers aspect also makes it difficult to see the combination technique as an instance of standard abstract domain constructions such as the direct or reduced product, or as a use of Hoare logic's conjunction rule.
– The shape analysis which will be strengthened explores the same abstract state space as the standard one would. That is, we do not explore the cartesian product of the shape and arithmetic state spaces. In this way the combined analysis treats the shape and arithmetic information independently (as in independent attribute analyses) except for the relations between shape and arithmetic information identified by the shape analysis as critical to memory- or assert-safety.
– Arithmetic refinement is performed only on-demand, when the standard shape analysis has failed to prove memory safety on its own.
– Because we track shape information at all program points, our analysis is able to verify properties such as memory-safety and absence of memory leaks.

## 2 Motivating Example

Consider the example code fragment in the left half of Figure 1. This program creates a list of length $n$ and then deletes it. Neither an arithmetic static analysis nor a traditional shape analysis alone can prove that `curr` is not equal to `NULL` at line 15. As we will see, our analysis is able to prove that this program is memory-safe.

Consider how a shape analysis without arithmetic support would treat this program. Using symbolic execution and widening, the analysis might find an invariant stating that, at location 4, *curr* is a pointer to a well-formed singly-linked and *NULL*-terminated list and $i$ is a pointer to a single heap cell. In separation logic, we would express this invariant as $\exists k, v.\ ls^k(curr, NULL) * i \mapsto v$ where *ls* is a recursively-defined list predicate and $k$ represents the length of the list. Note that the shape analysis has not attempted to infer any invariance properties of the integer values $k$ and $v$.

From this point the analysis might explore the path 4 → 12 → 13 → 14 → 15, obtaining

$$\exists k.\ ls^k(curr, NULL) \wedge j = 0 \wedge j < n \tag{1}$$

```
 1  List * curr = NULL;                   |  1  curr = 0;
 2  int i = malloc(sizeof(int));          |  2  skip;
 3  *i = 0;                               |  3  int v = 0;
                                          |     int k = 0;
 4  while(*i < n) {                       |  4  while(v < n) {
 5    t = (List*) malloc(sizeof(List));   |  5    t = nondet();
 6    t->next = curr;                     |  6    skip;
 7    t->data = addr;                     |  7    skip;
 8    addr += next_addr(addr);            |  8    addr += next_addr(addr);
 9    curr = t;                           |  9    curr = t;
10    *i = *i + 1;                        | 10    v = v + 1;
                                          |       k = k + 1;
11  }                                     | 11  }
12  free(i);                              | 12  skip;
13  int j = 0;                            | 13  int j = 0;
14  while(j < n) {                        | 14  while(j < n) {
15    t = curr->next;                     | 15    if(k > 0)
                                          |         b := nondet();
                                          |         t := b;
                                          |       else error();
16    free(curr);                         | 16    skip;
17    curr = t;                           | 17    curr = t;
18    j++;                                | 18    j++;
                                          |       k = k - 1;
19  }                                     | 19  }
```

**Fig. 1.** Left: Example showing motivation for combined shape and arithmetic reasoning. Right: Arithmetic counterexample program produced by the shape analysis.

At line 15, the program looks up the value in the *next* field of *curr*. But if the list is empty, then $curr = NULL$ and the lookup will fail. Because (1) does not imply that $curr \neq NULL$, this case cannot be ruled out and the analysis would report a potential violation of memory safety.

However, this case cannot actually arise due to the fact that the second loop frees only as many heap cells as the first loop allocates. To rule out this spurious counterexample, we need to strengthen the invariants associated with the loops, essentially discovering that the value stored in the heap cell at $i$ tracks the length of the list being created in the first loop and $j$ tracks the length of the unprocessed portion of the list in the second loop. Our algorithm achieves this by generating a *counterexample program* representing all paths that satisfy the shape formulas and could lead to the potential memory error.

The program we generate for this counterexample is given in the right half of Figure 1. We have numbered each line with the line number in the original program from which it is derived. Newly added commands are un-numbered. The counterexample program involves two new variables, $k$ and $v$, which represent the length of the list and the value pointed to by $i$, respectively.[1] New variables are added whenever the shape analysis encounters an integer value, such as the length of a list or the contents of an integer-valued heap cell.

Note that the control flow of the counterexample program is reminiscent of the control flow of the original program. The only difference here is that the counterexample program has an additional branch at location 15. This corresponds to a case split in the shape analysis—the memory access at location 15 in the original program is safe provided that $k$ (the length of the list) is greater than 0. Also note that heap commands have been replaced by purely arithmetic commands that approximate their effect on the arithmetic program's stack variables. Two examples of this are the command at location 5, where allocation is replaced by nondeterministic assignment, and the command at location 10, where the heap store command that updates the contents of $i$ is replaced by a command that updates the integer variable $v$.

Another unique aspect of our counterexample programs is that they may contain looping constructs. As such, they represent not just a single counterexample, but rather a set of counterexamples. Returning to the example in Figure 1, recall that the loop invariant at location 4 is $\exists k.\ ls^k(curr, NULL)$. To evaluate the memory safety of the command at location 15, we start with this invariant and compute postconditions along the path from 4 to 15. We then discover that the resulting postcondition is too weak to prove memory safety at location 15 and wish to generate a counterexample. Because the error state in the counterexample follows from the loop invariant at location 4, the counterexample can contain any number of unrollings of this loop. Rather than commit to a specific number and risk making overly specific conclusions based on the counterexample, we instead include a loop in the counterexample program. As we will see, this makes the set of paths through the counterexample program correspond to the full set

[1] The role of the third new variable, $b$, is more subtle. It arises due to expansion of a definition during theorem proving. This is discussed in detail in Section 4.1.

of abstract counterexamples. This ensures that the arithmetic tool generates a strengthening that rules out all spurious counterexamples (i.e. it is forced to discover a strengthening that is also a loop invariant) and is key to making the collaboration between the shape analysis and arithmetic analysis tool work.

Now let us look at this collaboration in more detail. While trying to prove that `error()` in the counterexample program (Figure 1) is not reachable, an arithmetic analysis tool such as Astrée[7], Blast [19], or ARMC [24] might prove the following arithmetic invariant at location 15: $k = n - j$. The soundness theorem for our system establishes that this invariant of the arithmetic counterexample program is also an invariant of the original program. As such, it is sound to conjoin this formula with our shape invariant at this location, obtaining $\exists k.\ ls^k(curr, NULL) \wedge k = n - j$. Note that the arithmetic invariant is conjoined inside the scope of the quantifier. This is sound because the variables we add to the counterexample program (such as $k$) correspond to the existentially quantified variables and their values correspond to the witnesses we used when proving those existential formulas. We formally prove soundness in Section 5.

Now, armed with the strengthened invariant, the shape analysis can rule out the false counterexample of *NULL*-pointer dereference at location 15. We will have the formula $ls^k(curr, NULL) \wedge k = n - j \wedge j < n$, from which we can derive $k > 0$—a sufficient condition for the safety of the memory access.

## 3 Preliminaries

Our commands include assignment ($e{:=}f$), heap load ($x{:=}[e]$), heap store ($[e]{:=}f$), allocation ($x{:=}\mathrm{alloc}()$), disposal ($\mathrm{free}(e)$), non-deterministic assignment (x := ?), and an `assume` command, which is used to model branch conditions. Note that brackets are used to indicate dereference. We use $C$ to denote the set of commands and the meta-variable $c$ to range over individual commands. The concrete semantics are standard (see [25]) and are omitted. We present only the concrete semantic domains and then move directly to a presentation of the abstract domain and its associated semantics.

The concrete semantic domain consists of pairs $(s, h)$, where $s$ is the *stack* and $h$ is the *heap*. Formally, the stack is simply a mapping from variables to their values, which are either integers or addresses.

$$Val \stackrel{\mathrm{def}}{=} Int \cup Addr$$

$$Stack \stackrel{\mathrm{def}}{=} Var \to Val$$

The heap is a finite partial function from non-null addresses to *records*, which are functions from a finite set of fields to values: $Record \stackrel{\mathrm{def}}{=} Field \to Val$, and $Heap \stackrel{\mathrm{def}}{=} (Addr - \{0\}) \stackrel{\mathrm{fin}}{\rightharpoonup} Record$. We also have a state **abort** which is used to indicate failure of a command. This may occur due to a failed assert statement or an attempt to dereference an address that is not in the domain of the heap.

Our analysis uses a fragment of separation logic [25] as an abstract representation of the contents of the stack and the heap. We have expressions for denoting addresses and records. Address expressions are simply variables or the constant

*NULL*, which denotes the null address. Integer expressions include variables and the standard arithmetic operations. Value expressions refer to expressions that may denote either integers or addresses. Record expressions are lists of field labels paired with value expressions.

$$\begin{array}{ll}
\textit{Address} & e, f, g ::= x \mid \mathit{NULL} \\
\textit{Integer Expressions} & m, n ::= x \mid i \mid v_1 + v_2 \mid v_1 - v_2 \mid \ldots \\
\textit{Value Expressions} & v, k ::= e \mid m \\
\textit{Record} & \rho ::= \mathit{label}\colon v, \rho \mid \epsilon
\end{array}$$

Our predicates are divided into *spatial* predicates, which describe the heap, and *pure* predicates, which describe the stack. The predicate **emp** denotes the empty heap, and $e \mapsto [l_1 : v_1, l_2 : v_2, \ldots, l_n : v_n]$ describes the heap consisting of a single heap cell at address $e$ that contains a record where field $l_1$ maps to value $v_1$, $l_2$ maps to $v_2$, etc. The atomic pure predicates include the standard arithmetic predicates ($<$, $\leq$, $=$, etc.) and equality and disequality over address expressions. Spatial formulas are built from conjunctions of atomic spatial predicates using the * connective from separation logic. Intuitively, $P * Q$ is satisfied when the domain of the heap described by $P$ is disjoint from that described by $Q$. Thus, $(e \mapsto \rho_1) * (f \mapsto \rho_2)$ implies that $e \neq f$.

We also allow existential quantification and adopt the convention that unmentioned fields are existentially quantified. That is, if a record always contains fields $s$ and $t$, we write $e \mapsto [s\colon v]$ to abbreviate $\exists z.\ e \mapsto [s\colon v, t\colon z]$.

From the atomic predicates we can inductively define predicates describing data structures, such as the following predicate for singly-linked list segments.

$$\mathit{ls}^k(e, f) \stackrel{\text{def}}{=} \big(k > 0 \wedge \exists x'.\ e \mapsto [n\colon x'] * \mathit{ls}^{k-1}(x', f)\big) \vee \ \big(\mathbf{emp} \wedge k = 0 \wedge e = f\big)$$

The length of the list is given by $k$, while $e$ denotes the address of the first cell (if the list is non-empty) and $f$ denotes the address stored in the "next" field ($n$) of the last cell in the list. If the list is empty, then $k = 0$ and $e = f$.

Our implementation actually uses a doubly-linked list predicate. However, in this paper we will use the simpler singly-linked list predicate in order to avoid letting the details of the shape analysis obscure the arithmetic refinement procedure, which is our main focus.

Our abstract states are drawn from the following grammar, where we use the notation $\vec{x}$ to represent a list of variables.

$$\begin{array}{ll}
\textit{Spatial Form} & \Sigma ::= e \mapsto \rho \mid \mathit{ls}^k(e, f) \mid \mathbf{emp} \mid S_1 * S_2 \\
\textit{Pure Form} & \Pi ::= x \mid e \leq f \mid e = f \mid \neg P \mid P_1 \wedge P_2 \\
\textit{Memory} & M ::= \exists \vec{x}.\ \Sigma \wedge \Pi \mid \top
\end{array}$$

The formula $\top$ is satisfied by all concrete states, including **abort**, and is used to indicate failure of a command. Elements of $\Pi$ are called *pure formulas*, while elements of $\Sigma$ are called *spatial formulas*. We take terms from $M$ as the elements of our abstract domain and refer to them as *abstract state formulas*. We will use the meta-variables $S$, $P$, and $Q$ to refer to such formulas.

In the left column of Figure 3, we give the postcondition rules for our commands. These are given as Hoare triples $\{P\}\ c\ \{Q\}$, where $P$ and $Q$ are abstract state formulas. To take the postcondition of state $S$ with respect to command $c$, we search for an $S'$ such that $S \Rightarrow S'$ and $\{S'\}\ c\ \{Q\}$ is an instance of the rule for $c$ in Figure 3. The formula $Q$ is then the postcondition of the command. If we cannot find such an $S'$, this corresponds to a failure to prove memory safety of command $c$ and the abstract postcondition is $\top$. For more on this process, see the discussion of the "unfold" rule in [22] and the section on "rearrangement rules" in [13].

## 4 Algorithm

A shape analysis based on separation logic, such as those in [22] and [13], will generate an *abstract transition system* (ATS), which is a finite representation of the reachable states of the program given as a transition system $(A)$ with states labeled by abstract state formulas. Such formulas are either formulas of separation logic or $\top$, which indicates a potential violation of memory safety. If a path from the initial state to $\top$ is found (a *counterexample* to memory safety), our algorithm translates this path into an arithmetic program $(\mathrm{Tr}(A))$. This arithmetic program is then analyzed to obtain strengthenings for the invariants discovered during shape analysis. The results of the arithmetic analysis are then combined with the shape analysis results to produce a more refined ATS $(\hat{A})$. A particular property of this combination is that if $\top$ can be shown to be unreachable in the arithmetic program, then the original program is memory safe.

**Definition 1.** *An **abstract transition system** is a tuple $(Q, L, \iota, \rightsquigarrow)$ where $Q$ is a set of states, $\iota \in Q$ is the start state, and $L\colon Q \to S$ is a function that labels each state with a separation logic formula describing the memory configurations associated with that state (or $\top$). The last component, $\rightsquigarrow$ is a labeled transition relation. The labels are either program commands (c) or an empty label ($\epsilon$). Thus, $\rightsquigarrow\ \subseteq Q \times (C \cup \{\epsilon\}) \times Q$. For convenience, if $t \in (C \cup \{\epsilon\})$ and $q, q' \in Q$, we will write $q \overset{t}{\rightsquigarrow} q'$ to abbreviate $(q, t, q') \in\ \rightsquigarrow$.*

We assume that quantified variables in the state labels are $\alpha$-renamed to be disjoint from the set of variables present in the commands labeling the edges. We will refer to the edges labeled with commands as *postcondition edges* and the edges labeled with $\epsilon$ as *weakening edges.* The reason for these names can be seen in the following definition of well-formedness, which we require of our ATSs.

**Definition 2.** *An ATS $(Q, L, \iota, \rightsquigarrow)$ is **well-formed** iff for all $q, q' \in Q$ and $c \in C$, i) $q \overset{c}{\rightsquigarrow} q'$ implies that $\{L(q)\}c\{L(q')\}$ is a valid separation logic triple and ii) $q \overset{\epsilon}{\rightsquigarrow} q'$ implies $(L(q) \vdash L(q'))$ is a valid separation logic entailment.*

This ensures that the annotations associated with the abstract states are consistent with the commands labeling the edges. That is, if $q \overset{c}{\rightsquigarrow} q'$ and $c$ terminates when executed from a state satisfying $L(q)$, then it terminates in a state satisfying $L(q')$. Well-formedness also ensures that the weakening edges are valid entailments. The algorithms defined in [22] and [13] automatically construct an abstract transition system that satisfies this condition.

In order to focus on the specifics of generating arithmetic programs from counterexamples, which is the main contribution of this paper, we assume that the abstract transition system has already been generated by running a separation logic based shape analysis on the input program. The interested reader can refer to [22] and [13] for details on how the ATS is generated.

An example of the abstract transition system that the shape analysis might generate is given in Figure 2. This ATS corresponds to the program discussed in Section 2. Dotted lines are used for weakening edges, while solid lines denote postcondition edges. We abbreviate `assume`$(e)$ as $a(e)$. Note that the shape analysis has discovered an invariant for the loop at control location 4, indicated by the cycle at the bottom of the second column of states.

At control location 15, the system splits based on the value of $k$, the length of the list. This is the one non-standard modification we make in our separation logic shape analysis. Such an analysis would ordinarily try to execute `curr := [curr.next]` at location 15 given the precondition $\exists k.\ ls^k(t, NULL)$. Since this precondition does not imply that the command is memory safe (the list could be empty), the analysis would simply conclude $\top$. Instead, our shape analysis will check to see if there is some condition under which the memory access would be safe. More precisely, our theorem prover internally performs case splits and if one of these cases results in safe execution, it returns this condition to the analysis. The analysis then splits based on this condition and continues exploring the safe branch (the unsafe branch remains labeled with $\top$). For our definition of lists, this condition is always a check on the length of the list. This is a key component of our technique as it makes explicit the way in which size information about data structures affects the safety of the program. It will then be the job of the arithmetic analysis tool to show that the unsafe branch is infeasible due to arithmetic constraints among the variables.

### 4.1 Generating Arithmetic Programs

The arithmetic program is generated by converting edges in the ATS to commands that do not reference the heap. This translation involves making use of the information about heap cells that the shape analysis has provided. For example, given the state $x \mapsto [data\colon y+2]$, we know that the command `z = [x.data]` will result in $z$ containing the value $y + 2$. We can achieve the same effect with the command `z = y + 2`, which does not reference the heap but instead exploits the fact that the shape analysis has determined the symbolic value for the contents of the data field of $x$.

The fact that our formulas can involve existential quantifiers makes the combination more expressive, and the translation more involved. Given the formula $\exists y.\ x \mapsto [data\colon y + 2]$, it is clearly no longer sound to replace the command `z = x->data` with the command `z := y + 2`. Since $y$ is not a program variable,

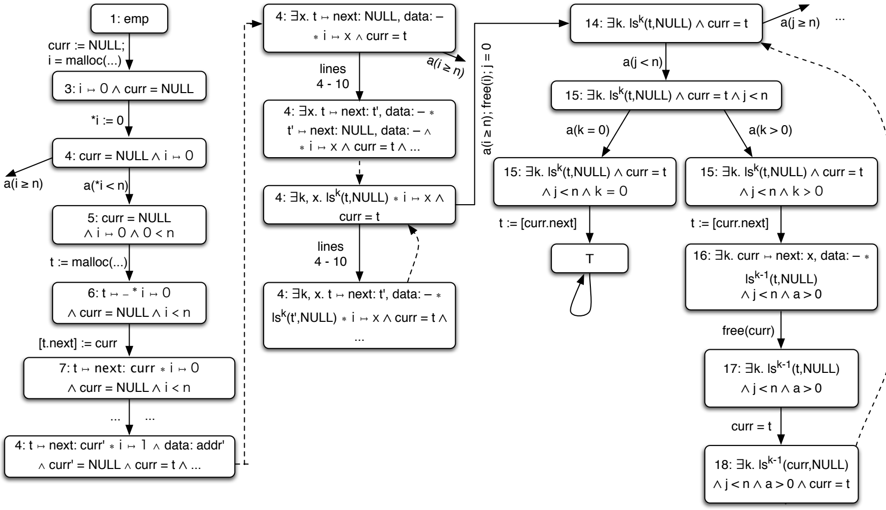

1: emp
curr := NULL;
i = malloc(...)
3: i ↦ 0 ∧ curr = NULL
*i := 0
4: curr = NULL ∧ i ↦ 0
a(i ≥ n)
a(*i < n)
5: curr = NULL ∧ i ↦ 0 ∧ 0 < n
t := malloc(...)
6: t ↦ – * i ↦ 0 ∧ curr = NULL ∧ i < n
[t.next] := curr
7: t ↦ next: curr * i ↦ 0 ∧ curr = NULL ∧ i < n
... ...
4: t ↦ next: curr' * i ↦ 1 ∧ data: addr' ∧ curr' = NULL ∧ curr = t ∧ ...
4: ∃x. t ↦ next: NULL, data: – * i ↦ x ∧ curr = t
a(i ≥ n)
lines 4 - 10
4: ∃x. t ↦ next: t', data: – * t' ↦ next: NULL, data: – ∧ * i ↦ x ∧ curr = t ∧ ...
4: ∃k, x. $ls^k$(t,NULL) * i ↦ x ∧ curr = t
lines 4 - 10
4: ∃k, x. t ↦ next: t', data: – * $ls^k$(t',NULL) * i ↦ x ∧ curr = t ∧ ...
a(i ≥ n); free(i); j = 0
14: ∃k. $ls^k$(t,NULL) ∧ curr = t
a(j ≥ n)
...
a(j < n)
15: ∃k. $ls^k$(t,NULL) ∧ curr = t ∧ j < n
a(k = 0)
a(k > 0)
15: ∃k. $ls^k$(t,NULL) ∧ curr = t ∧ j < n ∧ k = 0
15: ∃k. $ls^k$(t,NULL) ∧ curr = t ∧ j < n ∧ k > 0
t := [curr.next]
T
t := [curr.next]
16: ∃k. curr ↦ next: x, data: – * $ls^{k-1}$(t,NULL) ∧ j < n ∧ a > 0
free(curr)
17: ∃k. $ls^{k-1}$(t,NULL) ∧ j < n ∧ a > 0
curr = t
18: ∃k. $ls^{k-1}$(curr,NULL) ∧ j < n ∧ a > 0 ∧ curr = t

**Fig. 2.** Sample ATS after shape analysis

its value is not specified from the point of view of the arithmetic analysis tool. We must therefore ensure that the arithmetic program we generate contains a variable $y$, corresponding to the quantified variable in the formula and that in executions of the arithmetic program, $y$'s value is constrained in such a way that it satisfies the separation logic formulas we received from the shape analysis.

We can formalize this idea of using the arithmetic commands to enable reasoning about quantified variables with the following definition, which describes the properties the arithmetic command ($c'$ in the definition) must have.

**Definition 3.** *Let $q \overset{c}{\rightsquigarrow} q'$ be an edge in an ATS $(Q, L, \iota, \rightsquigarrow)$. Let $L(q) = \exists\vec{x}.\ P$ and $L(q') = \exists\vec{y}.\ Q$ be abstract state formulas. A command $c'$ is a* ***quantifier-free approximation*** *(QFA) of the edge $q \overset{c}{\rightsquigarrow} q'$ iff for any pure formulas $P'$ and $Q'$, the triple $\{P'\}\ c'\ \{Q'\}$ implies the triple $\{\exists\vec{x}.\ P \wedge P'\}\ c\ \{\exists\vec{y}.\ Q \wedge Q'\}$.*

That is, reasoning using $c'$ is an over-approximation of reasoning under the quantifier in the pre- and postconditions of $c$. In Section 5 on soundness, we show that such reasoning can be extended to the whole program by replacing each command in the original ATS with a quantifier-free approximation of that command and reasoning about the ATS thus obtained.

**Translating Postcondition Edges.** To find a purely arithmetic QFA for each of the heap-manipulating commands, let us first look at the rules that are used for adding postcondition edges to the ATS. These are given in the left column of Figure 3. They are presented as Hoare triples where the pre- and postconditions are abstract state formulas. We use the notation $S[x'/x]$ to mean $S$ with $x'$ substituted for $x$.

Note that the first three rules result in the abstract post-state having one more quantifier than the abstract pre-state: they each have the form

| Shape Analysis Postcondition Rule | | | Arith. Cmnd. |
|---|---|---|---|
| $\{\exists \vec{z}.\ S\}$ | $x{:=}\,E$ | $\{\exists x', \vec{z}.\ x = E[x'/x] \wedge S[x'/x]\}$ | $x'{:=}\,x;$ $x{:=}\,E[x'/x]$ |
| $\{\exists \vec{z}.\ S\}$ | $x{:=}\,?$ | $\{\exists x', \vec{z}.\ S[x'/x]\}$ | $x'{:=}\,x;\ \ x{:=}\,?$ |
| $\{\exists \vec{z}.\ S\}$ | $x{:=}\,\mathrm{alloc}()$ | $\{\exists x', \vec{z}.\ S[x'/x] * (x \mapsto [\ ])\}$ | $x'{:=}\,x;\ \ x{:=}\,?$ |
| $\{\exists \vec{z}.\ S * (E \mapsto [\rho, t\colon F])\}$ | $x{:=}\,[E.t]$ | $\{\exists x', \vec{z}.\ x = F[x'/x] \wedge (S * (E \mapsto [\rho, t\colon F]))[x'/x]\}$ | $x'{:=}\,x;$ $x{:=}\,F[x'/x]$ |
| $\{\exists \vec{z}.\ S * (E \mapsto [\rho])\}$ | $\mathrm{free}(E)$ | $\{\exists \vec{z}.\ S\}$ | $\epsilon$ |
| $\{\exists \vec{z}.\ S * (E \mapsto [\rho, t\colon G])\}$ | $[E]{:=}\,F$ | $\{\exists \vec{z}.\ S * (E \mapsto [\rho, t\colon F])\}$ | $\epsilon$ |
| $\{\exists \vec{z}.\ S\}$ | $\mathrm{assume}(P)$ | $\{\exists \vec{z}.\ S \wedge P\}$ | $\mathrm{assume}(P)$ |

**Fig. 3.** Rules for generating arithmetic commands from abstract postcondition edges

$\{\exists \vec{z}.\ S\}\ c\ \{\exists x, \vec{z}.\ S'\}$. Our goal is to find an arithmetic command $c'$ corresponding to the original command $c$, and to use $c'$ to reason about $c$. As such, we would like $c'$ to contain the new quantified variable. To do this in a way such that $c'$ is a QFA, we need $c'$ to record the witness for the existential in the postcondition. As an example, consider the command for assignment.

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

The variable $x'$ in the postcondition represents the old value of $x$. Thus, the value of $x$ before the assignment is the witness for $x'$ in the postcondition. We can record this fact using the sequence of commands $x'{:=}\,x;\ \ x{:=}\,E$. We use the same idea to handle the other two rules that add a quantifier.

Capturing the quantification in the new command is only part of the process. We must also over-approximate the effect of the command on the program variables. For commands like allocation ($x{:=}\,\mathrm{alloc}()$), the best we can do is replace this with the nondeterministic assignment $x{:=}\,?$. However, for lookup we can use the technique mentioned at the beginning of this section: if the precondition tells us that the $t$ field of cell $E$ contains the value $F$, we can replace $x{:=}\,[E.t]$ with $x{:=}\,F$ (and the precondition for lookup will always have this form).

The other heap commands (heap store and free) are replaced with no-ops. This may be surprising since these commands can have indirect effects on the values of integer variables in the abstract state formulas. Values stored in the heap can later be loaded into variables. This case is already handled by our rule for lookup, as can be seen by considering what happens when we translate the command sequence `[x.data] := y + 3; z := [x.data]` to arithmetic commands. The first command will be converted to a no-op. To translate the second command, we need to know its precondition. Supposing we start from the state $x \mapsto [\ ]$, the postcondition of the first command is $x \mapsto [\mathit{data}\colon y + 3]$. This means that the translation will convert the second command to $z{:=}\,y + 3$, which has the same effect on the program variable $z$ as the original commands. So indirect updates to program variables through the heap will be properly tracked.

Also, freeing memory cells can decrease the size of lists in the heap. To incorporate reasoning about the length of lists, we must talk about how we translate weakening edges in the ATS.

**Translating Weakening Edges.** Weakening edges are added by the shape analysis to the abstract transition system for two reasons. First, they are used to rewrite abstract states into a form to which we can apply one of the postcondition rules. For example, to execute `x := [a.next]` from the state

$$\exists k.\ ls^k(a, NULL) \wedge a \neq NULL$$

we must first notice that this formula implies

$$\exists y, k.\ a \mapsto [next\colon y] * ls^k(y, NULL) \wedge a \neq NULL$$

We can then apply the third postcondition rule to this state to get

$$\exists y, k.\ a \mapsto [next\colon y] * ls^k(y, NULL) \wedge a \neq NULL \wedge x = y$$

The other use of weakening edges is to show that certain formulas are invariant over executions of a loop. For example, suppose we start in a state

$$\exists k.\ ls^k(a, NULL)$$

And after executing some commands, reach the state

$$\exists x, k.\ a \mapsto [next\colon x] * ls^k(x, NULL)$$

If both these states are associated with the same program location, then we have found a loop invariant since the second formula implies the first. This fact is recorded in the ATS by connecting the second state to the first with a weakening edge.

In both cases, we need to record information about the quantified variables so that our arithmetic analysis can discover arithmetic relationships involving these quantified variables. As with postcondition edges, we do this by recording the witnesses for the quantified variables.

Recall that we have a weakening edge in the ATS only if $\exists\vec{x}.\ P \vdash \exists\vec{y}.\ Q$. Our goal then is to find an arithmetic command $c'$ such that for any $P', Q'$, if $\{P'\}c'\{Q'\}$ then $\exists\vec{x}.\ P \wedge P' \vdash \exists\vec{y}.\ Q \wedge Q'$. We generate such a $c'$ by analyzing the proof of entailment between $\exists\vec{x}.\ P$ and $\exists\vec{y}.\ Q$. As we are interested in tracking the values of existentially quantified variables, it is the rules for existential quantifiers that end up being important for generation of the arithmetic commands. In Figure 4 we present the standard rules for introduction and elimination of existential quantifiers, modified to produce the appropriate arithmetic commands. The full details of entailment for our fragment of separation logic are omitted for space reasons, but the system is similar to that described in [3].

The notation $P \vdash Q\langle c\rangle$ is used to mean that $P$ entails $Q$ and $c$ is the arithmetic command that is a quantifier-free approximation of this entailment. For existential elimination, we simply record the new constant that was introduced for reasoning about the quantified variable on the left. We also nondeterministically assign to the constant once we are done with it to ensure that it will not appear free in any invariants the arithmetic tool produces. For existential

$$\textsc{E-Elim}\quad \frac{\exists\vec{y}.\ P[a/x] \vdash \exists\vec{z}.\ Q \quad \langle c' \rangle}{\exists x, \vec{y}.\ P \vdash \exists\vec{z}.\ Q \quad \langle a{:=}\,x;\ \ c';\ \ a{:=}\,?\rangle} \qquad \textsc{E-Intro}\quad \frac{\exists\vec{y}.\ P \vdash \exists\vec{z}.\ Q[t/x] \quad \langle c' \rangle}{\exists\vec{y}.\ P \vdash \exists x, \vec{z}.\ Q \quad \langle x{:=}\,t;\ \ c'\rangle}$$

**Fig. 4.** Rules for generating arithmetic commands from proofs for weakening edges

introduction, we record the witness used to establish the existential formula on the right. We do this by having our entailment checker return a witness in addition to returning a yes/no answer to the entailment question. This is possible because the entailment procedure sometimes proves existentials constructively. When entailment is proved without finding a witness (*e.g.* as happens when unrolling an inductive definition with a quantified body), $t$ in the premise is a fresh logical constant, and so $x{:=}\,t$ is equivalent to $x{:=}\,?$.

As an example, suppose we want to generate arithmetic commands that model the entailment $\exists k.\ \mathit{ls}^k(a, \mathit{NULL}) \wedge a \neq \mathit{NULL} \vdash \exists x, k.\ a \mapsto [\mathit{next}\colon x] * \mathit{ls}^k(x, \mathit{NULL}) \wedge a \neq \mathit{NULL}$. We first introduce a new constant $b$ for the existential on the left, resulting in the formula $\mathit{ls}^b(a, \mathit{NULL}) \wedge a \neq \mathit{NULL}$ and the arithmetic command $b{:=}\,k$. We then unroll the list segment predicate according to the definition, obtaining $\exists x.\ a \mapsto [\mathit{next}\colon x] * \mathit{ls}^{b-1}(x, \mathit{NULL}) \wedge a \neq \mathit{NULL}$. Since $x$ arises due to the expansion of a definition, we use nondeterministic assignment in the generated command producing $x{:=}\,?$. We then apply existential elimination again, obtaining $a \mapsto [\mathit{next}\colon c] * \mathit{ls}^b(c, \mathit{NULL}) \wedge a \neq \mathit{NULL}$ and $c{:=}\,x$. Finally, we prove the formula on the right side of the entailment, obtaining witnesses for the existentially quantified variables $x$ (witness is $c$) and $k$ (witness is $b-1$). We then "forget" about the constants we added with the commands $b{:=}\,?;\ c{:=}\,?$. Thus, the full sequence of commands for this entailment is

```
b := k; x := ?; c := x; k := b - 1; b := ?; c := ?
```

The updates to $k$ here reflect the fact that, at this point in the execution, the length of the list predicate being tracked by the shape analysis has decreased in size by 1. Due to the commands `b := ?` and `c := ?`, any quantifier free invariant that holds after executing this sequence of commands will be expressed without reference to $b$ and $c$.

### 4.2 Precision

We can get a sense for the precision of this analysis by examining the places in which nondeterministic assignment is used to over-approximate a command. One such place is the rule for allocation. This should not concern us as the goal is to use these arithmetic programs to discover properties of the integer values involved in the program, whereas allocation returns a pointer value, which the shape analysis is already capable of reasoning about. We can use this observation to optimize our approach. If we keep track of type information we can ensure that we only generate arithmetic commands when those commands result in the update of integer-valued variables.

The other place where nondeterministic assignment occurs is in the rule for existential elimination when the entailment checker does not return a witness. This is actually the source of all imprecision in the arithmetic translation. It can happen that an integer value such as 3 is stored in a list element, resulting in the state

$$\exists k, d.\ x \mapsto [\mathit{data}\colon 3, \mathit{next}\colon k] * \mathit{ls}^d(k, \mathit{NULL})$$

If we then abstract this state to $\exists d.\ \mathit{ls}^d(x, \mathit{NULL})$, we lose the information about the value stored in the data field of the heap cell at $x$. If this field is accessed again, it will be assigned a nondeterministic value by the shape analysis. To remedy this would require a notion of refinement on the shape analysis side of the procedure. And indeed our technique would interact well with such a shape refinement system. One could interleave arithmetic refinement and shape refinement, calling one when the other fails to disprove a counterexample. We leave development of such a system for future work.

### 4.3 Combined Analysis

Using the translation of individual edges described above, we can define the translation of ATSs and the result of the combined analysis:

**Definition 4 (Translated arithmetic program).** *For an ATS* $A = (Q, L, i, \rightsquigarrow)$*, the* translated arithmetic program $\mathrm{Tr}(A) = (Q, L', i, \rightsquigarrow')$ *is an ATS defined such that if* $q \overset{c}{\rightsquigarrow} q'$ *and* $c'$ *is the arithmetic command associated with this edge, then we have* $q \overset{c'}{\rightsquigarrow'} q'$*.*

**Definition 5 (Combination).** *Given an ATS* $A = (Q, L, \iota, \rightsquigarrow)$ *and its well-formed translation* $\mathrm{Tr}(A) = (Q, L', \iota, \rightsquigarrow')$*, where* $L'(q)$ *is a pure formula for each* $q$*, the* combination *of* $A$ *and* $\mathrm{Tr}(A)$ *is defined to be the ATS* $\hat{A} = (Q, \hat{L}, \iota, \rightsquigarrow)$ *where if* $L(q) = \exists \vec{z}.\ S$ *and* $L'(q) = S'$ *then* $\hat{L}(q) = \exists \vec{z}.\ S \wedge S'$*.*

Note that **false** $\wedge \top$ is equivalent to **false**. So for an abstract state where the shape analysis obtained $\top$, indicating a potential safety violation, if an arithmetic analysis can prove the state is unreachable (has invariant **false**), then it is also unreachable in the combined analysis.

## 5 Soundness

The soundness result hinges on the fact that the translation for commands defined in Section 4 results in a quantifier-free approximation.

**Theorem 1.** *For each postcondition rule in Figure 3 the associated arithmetic command is a quantifier-free approximation of the original command.*

We also use the fact that the translation for weakening edges produces a quantifier-free approximation.

**Theorem 2.** *If* $\exists \vec{x}.\ P \vdash \exists \vec{y}.\ Q\ \ \langle c \rangle$ *and* $\{P'\}\ c\ \{Q'\}$ *then* $\exists \vec{x}.\ P \wedge P' \vdash \exists \vec{y}.\ Q \wedge Q'$*.*

Proofs of these theorems can be found in an expanded version of this paper [21]. Given these results, we can show that invariants discovered based on analyzing the arithmetic program can be soundly conjoined to the formulas labeling states in the ATS.

**Theorem 3 (Soundness).** *For an ATS $A$, suppose that we have run an arithmetic analysis on* $\mathrm{Tr}(A)$ *and obtained (pure) invariants at each program point. Then $\hat{A}$ is well-formed.*

*Proof.* This follows directly from the fact that the $c'$ commands are QFAs of the original edges. Let $q \overset{c}{\rightsquigarrow} q'$ be any edge in $\hat{A}$. Suppose $L(q) = \exists\vec{x}.\ P$ and $L(q') = \exists\vec{y}.\ Q$. Then $\hat{L}(q) = \exists\vec{x}.\ P \wedge L'(q)$ and $\hat{L}(q') = \exists\vec{y}.\ Q \wedge L'(q')$. We must show that the following triple holds

$$\{\exists\vec{x}.\ P \wedge L'(q)\}\ c\ \{\exists\vec{y}.\ Q \wedge L'(q')\}$$

Let $c'$ be the arithmetic command associated with this edge in $A'$. Since $\{L'(q)\}\ c'\ \{L'(q')\}$ and $c'$ is a QFA of $\{\exists\vec{x}.\ P\}\ c\ \{\exists\vec{y}.\ Q\}$, our goal follows immediately from the definition of QFA.

## 6 Experimental Results

We have developed a preliminary implementation of our analysis and tested it on a number of programs where memory safety depends on relationships between the lengths of the lists involved. For example, a function may depend on the fact that the result of filtering a list has length less than or equal to that of the original list. As arithmetic back-ends we have used OctAnal [23], Blast [19], and ARMC [24]. Preliminary results show two trends. First, there is no tool among those we tried that is strictly stronger than the others. That is, there is no tool among these three that is able to prove memory safety for all of our sample programs. However each program was able to be proven by some tool. In such cases, the ability to choose any arithmetic tool allows one to prove the greatest number of programs. Secondly, the performance characteristics of the tools are highly dependent on the type of input they are given. As our examples are all relatively small, OctAnal outperformed the tools based on model checking. However, for large programs that contain many arithmetic commands which are not relevant to proving memory safety, we would expect the relative performance of model checking tools to improve, as these tools only consider the variables needed to prove the property of interest. More experiments are necessary to fully explore the advantages and disadvantages of various arithmetic provers in the context of our combination procedure.

## 7 Related Work

The work presented here describes a way of lazily combining two abstract interpreters: the shape analysis produces abstracted versions of the input program for which an arithmetic analysis is then called. More eager combination approaches have been previously discussed in the literature (*e.g.* [11,17,18]).

Recent work [6] has described a method in which the TVLA [27] shape analysis is lazily combined with an arithmetic analysis based on Blast. This work reverses the strategy that we propose: they are lazily providing some additional spatial support for what is primarily an arithmetic analysis, whereas we are lazily providing additional arithmetic support for a shape analysis. Which approach is better depends on the program in question. Programs that are concerned primarily with integer calculations, but occasionally use a heap data structure may be better analyzed with the approach in [6]. Programs which have as their main function manipulation of heap data, or for which memory safety must be verified, would be better analyzed with our approach.

Another related approach is the shape analysis in [22], which uses predicate abstraction to retain facts about integer values during widening, but does not provide a predicate inference scheme. Thus, these predicates must be supplied by the user. Since our method uses a separate arithmetic tool to perform the refinement, we inherit any predicate inference that tool may perform.

Connections between shape and arithmetic reasoning are exploited throughout the literature (*e.g.* [1,15,10,8,29,14]). Also, people have looked at ways of combining abstract interpreters over different domains [11,17,18]. For example, one could imagine combining the shape analysis in [22] or [13] with an abstract interpretation over the domains of convex polyhedra [12] or octagons [23]. Our approach has the advantage of allowing the use of any of these abstract domains as well as arithmetic analyses that are not based on abstract interpretation. Furthermore, given the way in which information about quantified values is shared between the analyses, it is not clear that our approach can be seen as an instance of one of the standard constructions for combinations of abstract domains.

Other shape analyses are known to support arithmetic reasoning, but typically in only very limited ways that allow them to use naive arithmetic widening steps. For example, the shape analysis described in [4] provides a combined analysis that maintains arithmetic information. In this case the set of arithmetic variables in the abstract domain is extremely limited: each list-segment in the shape analysis invariant is associated with an arithmetic variable. Furthermore, only one inequality per variable is allowed, as the inequalties only occur between a variable and its “old version”. Given these restrictions, the widening operation in [4] can be naive in terms of its handling of arithmetic. Our refinement-based procedure uses arbitrary arithmetic analysis tools to strengthen the shape analysis invariant being inferred, meaning that we have access to the most sophisticated widening operations available. More arithmetic is supported in [9], but also with an aggressive widening since the arithmetic reasoning is targeted to within a loop body.

Another combination of shape and arithmetic is given in [26], which presents a means of reasoning about size properties of data structures tracked via a shape analysis based on reference counting and must-alias information.

A number of approaches based on combining a numerical analysis with a shape analysis based on shape graphs (such as [27]) have been explored. Examples include [16] and [28]. However ours is the first attempt to carry out such general arithmetic reasoning in a shape analysis where the abstract domain consists of separation logic formulas.

Our method makes use of a notion of *generalized path* (*i.e.* a path through the program where the number of unrollings through some loops are unspecified). Uses of this concept can be found elsewhere in the literature (*e.g.* [20,5]). In particular, our work can be seen as fitting nicely into the framework proposed in [5]. As in this work, we use a refinement procedure based upon analyzing generalized paths. However, our work is unique in that the paths arise due to a shape analysis based on abstract interpretation rather than a software model checker. Furthermore, the way in which quantifiers in the generalized path are expressed as variables in the translated path is not present in this other work.

## 8 Conclusion

Shape analyses are typically imprecise in their support for numerical reasoning. While an analysis that fully tracks correlations between shape and arithmetic information would typically be overkill, we often need a small amount of arithmetic information in shape analysis when arithmetic and spatial invariants interact. We have proposed a lazy method of combining a fixed shape analysis with an arbitrary arithmetic analysis. This method treats shape and arithmetic information independently except for key relationships identified by the shape analysis. Crucially, these relationships may be over values which are only present in the abstract states. When potentially spurious counterexamples are reported by our shape analysis, our method constructs a purely arithmetic program and uses available invariant inference engines as a form of refinement. This new adaptive analysis is useful when a proof of memory safety or `assert`-validity requires deep spatial reasoning with targeted arithmetic support.

## References

1. Armando, A., Benerecetti, M., Mantovani, J.: Model checking linear programs with arrays. Electr. Notes Theor. Comput. Sci. 144(3), 79–94 (2006)
2. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.: Automatic predicate abstraction of C programs. In: PLDI'2001: Programming Language Design and Implementation, vol. 36, pp. 203–213. ACM Press, New York (2001)
3. Berdine, J., Calcagno, C., O'Hearn, P.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, Springer, Heidelberg (2005)
4. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.: Automatic termination proofs for programs with shape-shifting heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
5. Beyer, D., Henzinger, T., Majumdar, R., Rybalchenko, A.: Path invarints. In: PLDI (2007)
6. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 532–546. Springer, Heidelberg (2006)

7. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI'2003: Programming Language Design and Implementation, pp. 196–207. ACM Press, New York (2003)
8. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with lists are counter automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
9. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, Springer, Heidelberg (2006)
10. Choi, Y., Rayadurgam, S., Heimdahl, M.P.: Automatic abstraction for model checking software systems with interrelated numeric constraints. In: Proc. of ESEC/FSE, pp. 164–174. ACM Press, New York (2001)
11. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL'1979: Principles of Programming Languages, pp. 269–282. ACM Press, New York (1979)
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL (1978)
13. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, Springer, Heidelberg (2006)
14. Dor, N., Rodeh, M., Sagiv, M.: Cssv: towards a realistic tool for statically detecting all buffer overflows in c. In: PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, New York, Ny, USA, pp. 155–167. ACM Press, New York, USA (2003)
15. Flanagan, C.: Software model checking via iterative abstraction refinement of constraint logic queries. In: CP+CV'04 (2004)
16. Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M.: Numeric domains with summarized dimensions. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, Springer, Heidelberg (2004)
17. Gulwani, S., Tiwari, A.: Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In: Sestoft, P. (ed.) ESOP 2006 and ETAPS 2006. LNCS, vol. 3924, pp. 279–293. Springer, Heidelberg (2006)
18. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: Ball, T. (ed.) ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI 2006, ACM Press, New York (2006)
19. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL'2002: Principles of Programming Languages, pp. 58–70. ACM Press, New York (2002)
20. Kroening, D., Weissenbacher, G.: Counterexamples with loops for predicate abstraction. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
21. Magill, S., Berdine, J., Clarke, E., Cook, B.: Arithmetic strengthening for shape analysis. Technical Report CMU-CS-07-135, Carnegie Mellon University (2007)
22. Magill, S., Nanevski, A., Clarke, E., Lee, P.: Inferring invariants in separation logic for imperative list-processing programs. In: SPACE 2006: Third Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (2006)
23. Miné, A.: The Octagon abstract domain. Higher-Order and Symbolic Computation (to appear)

24. Podelski, A., Rybalchenko, A.: ARMC: the logical choice for software model checking with abstraction refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, Springer, Heidelberg (2006)
25. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE Computer Society Press, Los Alamitos (2002)
26. Rugina, R.: Quantitative shape analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, Springer, Heidelberg (2004)
27. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: TOPLAS (2002)
28. Yavuz-Kahveci, T., Bultan, T.: Automated verification of concurrent linked lists with counters. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, Springer, Heidelberg (2002)
29. Zhang, T., Sipma, H.B., Manna, Z.: Decision procedures for queues with integer constraints. In: Ramanujam, R., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 225–237. Springer, Heidelberg (2005)

# Astrée: From Research to Industry

David Delmas and Jean Souyris

Airbus France S.A.S.
316, route de Bayonne
31060 TOULOUSE Cedex 9, France
{David.Delmas,Jean.Souyris}@Airbus.com

**Abstract.** Airbus has started introducing abstract interpretation based static analysers into the verification process of some of its avionics software products. Industrial constraints require any such tool to be extremely precise, which can only be achieved after a twofold specialisation process: first, it must be designed to verify a class of properties for a family of programs efficiently; second, it must be parametric enough for the user to be able to fine tune the analysis of any particular program of the family. This implies a close cooperation between the tool-providers and the end-users. Astrée is such a static analyser: it produces only a small number of false alarms when attempting to prove the absence of run-time errors in control/command programs written in C, and provides the user with enough options and directives to help reduce this number down to zero. Its specialisation process has been reported in several scientific papers, such as [1] and [2]. Through the description of analyses performed with Astrée on industrial programs, we give an overview of the false alarm reduction process from an engineering point of view, and sketch a possible customer-supplier relationship model for the emerging market for static analysers.

**Keywords:** avionics software, verification, abstract interpretation, static analysis, run-time errors, Astrée.

## 1 Introduction

Verification activities are responsible for a large part of the overall costs of avionics software developments. Considering the steady increase of the size and complexity of this kind of software, classical Validation and Verification processes, based on massive testing campaigns and complementary intellectual analyses, hardly scale up within reasonable costs. Therefore, Airbus has decided to introduce formal proof techniques providing product-based assurance into its own verification processes.

Available formal methods include model checking, theorem proving and abstract interpretation based static analysis ([3], [4], [5]). The items to be verified being final products, i.e. source or binary code, model checking is not considered relevant. Some theorem proving techniques have been successfully introduced to verify limited software subsets. However, to prove properties of complete real-size programs with these techniques does not seem to be within the reach of software engineers yet.

On the other hand, abstract interpretation based static analysers aiming at proving specific properties on complete programs have been shown to scale to industrial

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 437 – 451, 2007.

safety-critical programs. Among these properties, one is to quote worst-case execution time assessment ([7]), stack analysis, accuracy of floating-point computations ([8]), absence of run-time errors, etc. Moreover, these analysers are automatic, which is obviously a requirement for industrial use.

Yet, however precise any such tool may be, a first run of it on a real-size industrial software will typically produce at least a few false alarms. For safety and industrial reasons, this number of false alarms should be as small as possible, and an engineering user should be able to be reduce it down to zero by a new fine tuned analysis. Indeed, the fewer false alarms are produced, the fewer costly, time-consuming and error-prone complementary intellectual analyses are necessary. Hence the need for both specialised and parametric tools outputting comprehensible diagnoses, which can only be achieved through a close cooperation between the tool-providers and the end-users.

The Astrée static analyser has proved to meet these requirements. In this paper, we first give an overview of this tool and its specialisation process. Then, we describe the analysis process: how we run the tool, how we analyse the resulting alarms, and how we tune the parameters of the tool to reduce the number of false alarms. Next, we illustrate the alarm reduction process with a report on the analysis of a real-size industrial control/command program, and an example of alarm analysis for another avionics program. Finally, we assess the analysis results, and state Airbus's position on the perspectives for a tool such as Astrée within a possible new kind of customer-supplier relationship.

## 2 The Astrée Analyser

Astrée is a parametric Abstract Interpretation based static analyser that aims at proving the absence of RTE (Run-Time Errors) in programs written in C.

The underlying notion has been defined in several papers, such as [2, §2]: "*The absence of runtime errors is the implicit specification that there is no violation of the C norm (e.g., array index of bounds), no implementation-specific undefined behaviours (e.g., floating-point division by zero), no violation of the programming guidelines (e.g., arithmetic operators on short variables should not overflow the range* `[-32768,32767]` *although, on the specific platform, the result can be well-defined through modular arithmetics).*"

As explained in [1, §3.1], this tool results from the refinement of a more general-purpose analyser. It has been specialised in order to analyse synchronous control/command programs very precisely, thanks to specific iterator and abstract domains described in [1]. This is the result of the "per program family specialisation process". Furthermore, the parametric nature of Astrée makes it possible for the user to specialise it for any particular program within the family.

## 3 The Alarm Reduction Process

Even for a program belonging to the family of synchronous control/command programs, the first run of Astrée will usually produce false alarms to be further investigated by the industrial user. The user must tune the tool parameters to improve the precision of the analysis for a particular program. This final "per program specialisation process" matches the adaptation by parameterisation described in [1, §3.2].

### 3.1 The Need for Full Alarm Investigation

We do not use Astrée to search for possible run-time errors; we use it in order to prove that no run-time error can ever occur. As a consequence, every single alarm has to be investigated.

Besides, every time Astrée signals an alarm, it assumes the execution of the analysed program to stop whenever the precondition of the alarm is satisfied, because the program behaviour is undefined in case of error, e.g. an out-of-bounds array assignment might destroy the code[1]. Thus, any satisfiable alarm condition may "hide" more alarms.

Let us give a simple example with variable `i` of type `int` in interval `[0,10]`:

```
1       int t[4];
2       int x = 1;
3       int y;
4       t[i] = 0;
5       y = 1/x;
```

Astrée reports a warning on line 4 (invalid dereference), but not on line 5. However, executing instruction 4 with `i>3` will typically overwrite the stack, e.g. set variable `x` to `0`, so that instruction 7 may produce a division by zero. Since the execution is assumed to stop whenever `i>3` on line 4, Astrée assumes `i` to be in interval `[0,3]` from line 4.

That is the reason why exhaustive alarm analysis is required: every false alarm should disappear by means of a more precise automated analysis, or, failing that, be proved by the user to be impossible in the real environment of the program.

### 3.2 How to Read an Alarm Message

The following alarm will be discussed later in this paper:

```
P3_A_3.c:781.212-228::

[call#APPLICATION_ENTRY@449:loop@466>=4:call#SEQ_C4_10_
P@673:call#P_3_A_3_P@118:]:

WARN: float arithmetic range [-inf, inf] not included
in [-3.40282e+38, 3.40282e+38]
```

Let us explain how this message reads. Astrée warns that some simple precision floating-point computation may yield a result that cannot be represented in the type

[1] For a complete explanation, refer to [6, §4.1].

`float`. It points precisely to the operation that may cause such a run-time error: line 781 of the (pre-processed) file P3_A_3.c, between columns 212 and 228[2]:

```
_R1= PADN10 - X3A3Z15 ;
```

where variables `_R1`, `PADN10` and `X3A3Z15` have type `float`.

All other information describe the context of the alarm. The analysis entry point[3] is the `APPLICATION_ENTRY` function, defined on line 449 of some file. This function contains a loop on line 466. From the fourth loop iteration, at least in the abstract semantics computed by the tool, there exists an execution trace such that:

- function `SEQ_C4_10_P` is called on line 673;
- `SEQ_C4_10_P`, defined in some file, calls function P_3_A_3_P on line 118 of this file;
- the result of the floating-point subtraction of the operands `PADN10` and `X3A3Z15` does not range in `[-3.40282e+38, 3.40282e+38]`.

Of course, this does not necessarily mean there exists such an erroneous execution in the concrete semantics of the program: one is now to address this issue via a dedicated method.

### 3.3 Dealing with Alarm Investigation

As explained above, every alarm message refers to a program location in the preprocessed code. It is usually useful to get back to the corresponding source code, to obtain readable context information.

When Astrée fails to prove an operation free from run-time errors, it outputs an alarm message, together with a brief explanation of the reason why the alarm was raised. Most such alarm conditions are expressed in terms of intervals. To investigate them, one makes use of the global invariant of the most external loop of the program, which is available in the Astrée log file (provided the `--dump-invariants` analysis option is set). Considering every global variable processed by the operation pointed to by an alarm, one may extract the corresponding interval, which is a sound over-approximation of the range of this variable[4]. The user may also use the `__ASTREE_log_vars((`$V_1$`,...,`$V_n$`));` directive when the ranges of local variables are needed.

Then, we have to go backwards in the program data-flow, in order to get to the roots of the alarm: either a bug or insufficient precision of the automated analysis. This activity can be quite time-consuming. However, it can be made easier for a control/command program that has been specified in some graphical stream language such as SAO, SCADE™ or Simulink™, especially if most intermediate variables are declared global. The engineering user can indeed label every arrow representing a

[2] Line numbers start from 1, whereas column numbers start from 0.

[3] The user provides Astrée with an entry point for the analysis, by means of the `--exec-fn` option. Usually, this is the entry point of the program.

[4] If the main loop is unrolled N times (to improve precision), the N first values of variables are not included in the global invariant. The `__ASTREE_log_vars((`$V_1$`,...,`$V_n$`));` directive is needed to have Astrée output these values. However, the global invariant is enough to deal with alarms occurring after the N$^{th}$ iteration.

global variable with an interval, going backwards from the alarm location. The origin of the problem is usually found when some abrupt inexplicable increase in variable ranges is detected.

At this point, we know whether the alarm originated in some local code with limited effect or in some definite specialised operator (i.e., function or macro-function). Indeed, an efficient approach is first to concentrate on alarms in operators that are used frequently in the program, especially if several alarms with different stack contexts point to the same operators: such alarms will usually affect the analysis of the calling functions, thus raising more alarms. For control/command programs with a fairly linear call-graph, it can be also quite profitable to pick alarms originating early in the data-flow first. To get rid of such alarms may help eliminate other alarms originating later in the data-flow.

Once we have found the roots of the alarm, we will usually need to extract a reduced example to analyse it. Therefore, we:

- write a small program containing the code at stake;
- build a new configuration file for this example, where the input variables `V` are declared `volatile` by means of the `__ASTREE_volatile_input((V [min, max]));` directive. The variable bounds are extracted from the global invariant computed by Astrée on the complete program;
- run Astrée on the reduced example (which takes far less time than on a complete program).

Such a process is not necessarily conservative in terms of RTE detection. Indeed, as the abstract operators implemented in Astrée are not monotonic, an alarm raised when analysing the complete program may not be raised when analysing the reduced example. In this case, this suggests (though does not prove) that the alarm under investigation is probably false, or that the reduced example is not an actual slice of the complete program with respect to the program point pointed to by the alarm.

However, this hardly ever happens in practice: every alarm raised on the complete program will usually be raised on the reduced example as well. Furthermore, it is much easier to experiment with the reduced example:

- adding directives in the source to help Astrée increase the precision of the analysis;
- tuning the list of analysis options;
- changing the parameters of the example itself to better understand the cause of the alarm.

Once a satisfactory solution has been found on reduced examples, it is re-injected into the analysis of the complete program: in most cases, the number of alarms decreases.

## 4 Verifying a Control/Command Program with Astrée

Let us illustrate this alarm reduction process for a periodic synchronous control/command program developed at Airbus. Most of its C source code is generated automatically from a higher-level synchronous data-flow specification. Most generated

C functions are essentially sequences of calls of macro-functions coded by hand. Like in [1, §4], it has the following overall form:

```
declare volatile input, state and output variables;
initialise state variables;
loop forever
    read volatile input variables,
    compute output and state variables,
    write to volatile output variables;
    wait for next clock tick;
end loop
```

This program is composed of about 200,000 lines of (pre-processed) C code processing over 10,000 global variables. Its control-flow depends on many state variables. It performs massive floating-point computations and contains digital filters.

Although an upper bound of the number of iterations of the main loop is provided by the user, all these features make precise automatic analysis (taking rounding errors into account) a grand challenge. A general-purpose analyser would not be suitable.

Fortunately, Astrée has been specialised in order to deal with this type of programs: only the last step in specialisation (fine tuning by the user) has to be carried out. The automated analyses are being run on a 2.6 GHz, 16 Gb RAM PC. Each analysis of the complete program takes about 6 hours.

### First Analysis

#### Program Preparation

The program is being prepared in the following way:

- some assembly functions are recoded in C or removed;
- compiler built-in functions are redefined:

```
double fabs(double x) {
    if (x>=0.) return x; else return (-x);
}
double sin(double x) {
    double y;
    __ASTREE_known_fact((y>=-1.0));
    __ASTREE_known_fact((y<=1.0));
    return y;
}
double cos(double x) {
    double y;
    __ASTREE_known_fact((y>=-1.0));
    __ASTREE_known_fact((y<=1.0));
    return y;
  }
  volatile void waitforinterrupt(void) {
    __ASTREE_wait_for_clock(());
}
```

In this way, we provide Astrée with a model of external functions. On the one hand, the `ASTREE_known_fact((...));` directive helps Astrée bound the values computed by trigonometric functions. On the other hand, the `__ASTREE_wait_for_clock(());` directive delimits the code executed on each iteration of the main loop. Its counterpart is the `__ASTREE_max_clock((3600000));` directive in the analysis configuration file, which provides Astrée with an upper bound of the number of iterations of this loop.

Finally, the analysis configuration file contains `__ASTREE_volatile_input((V [min, max]));` directives describing the ranges of all the volatile inputs of the program.

### Analysis Options

**Table 1.** List of options

| Option | Meaning |
|---|---|
| `--config-sem prog.config` | Analysis configuration file. |
| `--exec-fn APPLICATION_ENTRY` | Entry point of the program. |
| `--inner-unroll 15` | Inner loops[5] are unrolled at most 15 times (to improve precision). |
| `--dump-invariants` | Prints the invariant of the most external loop of the program, i.e. the ranges of all global variables. |

### Results

Under the above conditions, this first analysis produces 467 alarms.

Let us take a closer look at the three following messages, the first of which has been described earlier:

```
P3_A_3.c:781.212-228::
[call#APPLICATION_ENTRY@449:loop@466>=4:call#SEQ_C4_10_
P@673:call#P_3_A_3_P@118:]:
WARN: float arithmetic range [-inf, inf] not included
in [-3.40282e+38, 3.40282e+38]
```

[5] The main loop is unrolled 3 times (default).

```
P3_A_3.c:781.355-362::

[call#APPLICATION_ENTRY@449:loop@466>=4:call#SEQ_C4_10_
P@673:call#P_3_A_3_P@118:if@781=true:]:

WARN: float arithmetic range [-inf, inf] not included
in [-3.40282e+38, 3.40282e+38]

P3_A_3.c:781.409-416::

[call#APPLICATION_ENTRY@449:loop@466>=4:call#SEQ_C4_10_
P@673:call#P_3_A_3_P@118:if@781=true:]:

WARN: float arithmetic range [-inf, inf] not included
in [-3.40282e+38, 3.40282e+38]
```

Floating-point overflow is being suspected. Let us show line 781 of the pre-processed P3_A_3.c file:

```
{static NUM _R1;static INT _R2;static BOO _R3; if (
BLBPO ) { _R1=0; _R2=0; if ( B3A3Z09 )  X3A3Z15 =
PADN10 ; else  X3A3Z15 = SYNC_11_E2 ; } else { if (
B3A3Z09  ^ _R3) { if ( B3A3Z09 ) { _R2= SYNC_11_E7 ;
_R1= PADN10 - X3A3Z15 ; } else { _R2= SYNC_11_E4 ; _R1=
SYNC_11_E2 - X3A3Z15 ; } } else { if (_R2>0) _R2=_R2-1;
if ( B3A3Z09 )  X3A3Z15 =( PADN10 -(_R1*_R2/ SYNC_11_E7
)); else  X3A3Z15 =( SYNC_11_E2 -(_R1*_R2/ SYNC_11_E4
)); } } _R3= B3A3Z09 ;}
```

We emphasize the three program locations using bold type. Looking up in the source file, we can see this is an expansion of macro-function SYNC:

```
SYNC(11,PADN10,SYNC_11_E2,B3A3Z09,SYNC_11_E4,BLBPO,SYNC
_11_E7,X3A3Z15)
```

From constant definitions and global variable ranges, we can find the values or intervals of every variable occurring in the computation.

No code is generated for “11”, a macro-function occurrence number. PADN10 is an intermediate variable used in several contexts, so its global interval is of no use. Yet, looking at the source code, we easily notice that this variable is no more than a copy of global variable X3A3Z01, the range of which has been computed by Astrée :

```
X3A3Z01 in [-1e+06, 1.41851e+06]
```

The SYNC_11_E2 float constant has value 0. All other inputs of the SYNC macro-function are integer constants (with value 17) or Booleans. Astrée has also output an interval for the result of the macro-function:

```
X3A3Z15 in [-3.40282e+38, 3.40282e+38]
```

Unsurprisingly, considering overflow is suspected, that is the largest possible range for a simple-precision floating-point number.

In order to analyse these alarms, we may wonder why the X3A3Z01 input has so large a range. Looking a few lines backwards in the data-flow, we notice its interval depends upon the analysis by Astrée of another occurrence of the SYNC macro-function:

```
SYNC(14,X3A3Z09,SYNC_14_E2,BAPRO2U,SYNC_14_E4,BLBPO,SYN
C_14_E7,X3A3Z01)

INV(1,BLBPO,PADB12)

ET(1,PADB12,BIMPACC,PADB11)

MEM_N(19,X3A3Z01,PADB11,PADN10)

CONF1_I(11,BLSOL,CONF1_11_E2,CONF1_11_E3,BLBPO,PADB15)

INV(2,PADB15,B3A3Z09)

SYNC(11,PADN10,SYNC_11_E2,B3A3Z09,SYNC_11_E4,BLBPO,SYNC
_11_E7,X3A3Z15)
```

We can look up the range of the input of this first SYNC in the global invariant:

```
X3A3Z09 in [-4966.87, 6738.46]
```

The analysis of this first SYNC has multiplied the ranges between the X3A3Z09 input and the X3A3Z01 output by a factor of 200. The factor is even higher for the second SYNC. Building more SYNC-based reduced examples, we easily convince ourselves that the analysis of this macro-function causes variable ranges to blow up. The larger the input range, the larger the factor. As a consequence, several occurrences of it in the data-flow will eventually cause alarms, hence maximal simple-precision range, hence more alarms when the outputs are used elsewhere.

For once, Astrée does not implement a dedicated abstract domain to handle this type of code. There is no way the user can make the analysis more precise. This is where support from the tool-provider is needed.

From the reduced example extracted by Airbus, the Astrée development team found out that the frequency of widening steps was too high for this macro-function to be analysed precisely enough. They delivered a new version of the tool implementing new options, for the user to be able to tune widening parameters. In particular, the new `--fewer-widening-steps-in-intervals <k>` option makes Astrée widen unstable interval constraints k times less often. On the reduced examples, all alarms disappear with k=2.

### 4.1 Improving the Precision of the Analysis

The `--fewer-widening-steps-in-intervals 2` option is being added to the list of analysis options. All SYNC-related alarms disappear, and we get 327 remaining alarms.

We notice that many calling contexts of the widely used linear two-variable interpolation function G_P give rise to alarms within the source code of this function. Here is an example:

```
g.c:200.8-55::

[call#APPLICATION_ENTRY@449:loop@466>=4:call#SEQ_C1_P@7
11:call#P_2_7_1_P@360:call#G_P@977:if@132=false:if@137=
true:if@165=false:if@169=false:loop@177=2:]:

WARN: float division by zero [0, 45]
```

To understand the problem and be able to tune the analysis parameters, one is to build a reduced example from function `P_2_7_1_P`. The following code is being extracted from the original function:

```
void P_2_7_1_P () {

    PADN13 = fabs(DQM);

    PADN12 = fabs(PHI1F);

    X271Z14 = G_P(PADN13, PADN12, G_50Z_C1, G_50Z_C2, &
    G_50Z_C3 [0][0], & G_50Z_C4 [0][0], ((sizeof(
    G_50Z_C1 )/sizeof(float))-1), (sizeof( G_50Z_C2
    )/sizeof(float))-1);

}
```

where:

- `fabs` returns the module of a floating-point number;
- `DQM`, `PHI1F`, `PADN13`, `PADN12` and `X271Z14` are floating-point numbers;
- `G_50Z_C1`, `G_50Z_C2`, `G_50Z_C3` and `G_50Z_C4` are constant interpolation tables.

`DQM` and `PHI1F` are declared as volatile inputs in the analysis configuration file. Their ranges are extracted from the global invariant computed by Astrée on the full program:

```
DQM in [-37.5559, 37.5559]

PHI1F in [-199.22, 199.22]
```

On this reduced example, we get the same alarms as on the full program. All of them suspect an overflow or a division by zero in the last instruction of the `G_P` function:

```
return(Z2*(Y2-C2[R3])+Z1*(C2[G2]-Y2))/(C2[G2]-C2[R3]);
```

However, when reading the code of `G_P`, one notices that `G2=R3+1` always holds at this point. Moreover, in this reduced example, the interpolation table `G_50Z_C2` is such that `G_50Z_C2[i+1]-G_50Z_C2[i]>1` for any index `i`. Hence, these alarms are false alarms; we must now tune the analysis to get rid of them.

To do so, we have to make Astrée perform a separate analysis for every possible value of `R3`, so it can check no RTE can possibly happen on this code. The way to do so is to ask for a local partitioning on `R3` values between:

- the first program point after which `R3` is no longer written;
- the first program point after which `R3` is no longer read.

Let us implement this, using Astrée partitioning directives:

```
__ASTREE_partition_begin((R3));

G2=R3+1;

Z1=(X1-C1[R2])*(*(C4+(TAILLE_X)*R3+R2)) +
(*(C3+(TAILLE_X+1)*R3+R2));

Z2=(X1-C1[R2])*(*(C4+(TAILLE_X)*G2+R2)) +
(*(C3+(TAILLE_X+1)*G2+R2));

return(Z2*(Y2-C2[R3])+Z1*(C2[G2]-Y2))/(C2[G2]-C2[R3]);

__ASTREE_partition_merge(());
```

This hint makes the alarms disappear on the reduced example.

### 4.2 An Even More Precise Analysis

The analysis of the whole program is being re-launched after the partitioning directives have been inserted in the G_P function. All alarms within the G_P function disappear, and many alarms depending directly or indirectly on variables written after a call of function G_P disappear as well: the overall number of alarms boils down to 11.

Here is one of them:

```
PB_9_6.c:610.214-254::

[call#APPLICATION_ENTRY@449:loop@466>=4:call#SEQ_C3_2_P
@501:call#P_B_9_6_P@304:]:

WARN: float division by zero [0, 131070]
```

This alarm occurs within the code of the EANCAL_ANI6_0 macro-function. We use bold type to emphasize the program location which is referred to:

```
#define EANCAL_ANI6_0(NN,_S1) {\

...

if ((_REG_ANI6_PM1 < 0x19) || (_REG_ANI6_PM2 < 0x19))\

  {\

     BOVFANI6BIS0 = TRUE;\

     _S1=9216.0;\

  }\

else\

  {\

     BOVFANI6BIS0 = FALSE;\

     _S1=460800.0/(_REG_ANI6_PM1 + _REG_ANI6_PM2);\

  }\

};
```

where variables `_REG_ANI6_PM1` and `_REG_ANI6_PM2` of type `unsigned int` are declared volatile inputs in the configuration file of the analysis. This is obviously not a false alarm.

### 4.3 Results

On this control/command program, it has been possible for a non-expert user from industry to reduce the number of alarms down to zero.

## 5 Verifying Another Kind of Avionics Programs with Astrée

We will now give an example of alarm analysis on another synchronous program, where the need for specialisation is obvious. This program is not quite a control/command program. It lies on the boundary of the family of programs for which Astrée has been specialised. Nevertheless, the analyser is still precise on this program, raising few false alarms.

This avionics software product is meant to format data from input media to output media. It is composed of basic functions, and its control flow is defined by constant configuration tables. Unlike the previous program, it performs very limited floating-point computations, but processes many structured data types.

### 5.1 The Alarm

The alarm message to be further investigated is the following:

```
mess_conv.c:1058.29-85::

[call#main@8483:call#SQF_Se_Gateway@8501:loop@591=1:cal
l#XMM_Se_Message@612:call#XMC_Se_ReceiveUnrefreshMess@8
98:loop@1046>=2:]:

WARN: unsigned int->unnamed enum conversion range [0,
4294967295] not included in [0, 66]
```

The alarm occurs on line 1058 of the pre-processed `mess_conv.c` file, which we emphasize in bold type below:

```
1: if (IdFctConv < GST_Ct_T_STRUCT_GW_SIZE.NB_GW) {
2:   P_ID_Fct_Conv = (const XMT_Ts_Messages *)
                             &XMC_Ct_T_TABLE_GW[IdFctConv];
3:   Nb_Conv = P_ID_Fct_Conv->Conv_Number;
4:   Index_1ere_Conv = P_ID_Fct_Conv->Begin_List_Index;
5:   if ((Index_1ere_Conv < GST_Ct_T_STRUCT_GW_SIZE.NB_GW_LIST)
        && ((Index_1ere_Conv + (TCD_Td_uInt32) Nb_Conv)
           <= GST_Ct_T_STRUCT_GW_SIZE.NB_GW_LIST)) {
6:     for (cpt_nb_conv = 0;
        cpt_nb_conv < (TCD_Td_uInt32) Nb_Conv;
        cpt_nb_conv++) {
7:        Index_Conv = Index_1ere_Conv + cpt_nb_conv;
8:        P_List_Index_Conv =
             (const XMT_Ts_FunctionsListMessages *)
             &XMC_Ct_T_TABLE_GW_LIST[Index_Conv];
9:        Function_Id =(TED_Te_FunctionName)
                        P_List_Index_Conv->GW_Name_Function;
```

## 5.2 Analysis

Just before this piece of code, the abstract value of `IdFctConv` is `[0, 51]`. Consequently, the abstract value of the `P_ID_Fct_Conv` pointer after instruction 2 is an interval containing more than one value.

It follows that the abstract value of `Nb_Conv` is an interval: `[2, 342]`. Indeed, its lower bound is the minimum value of the `Conv_Number` field for elements of the `XMC_Ct_T_TABLE_GW[]` array with indexes ranging from 0 to 51, which is actually 2. Its upper bound is the maximum value of the same field in the same array slice, which is in fact 342.

Similarly, `Index_1ere_Conv` ranges in `[0, 1117]`.

Let us now consider instruction 6 (the for loop). The loop test expression is `cpt_nb_conv < (TCD_Td_uInt32) Nb_Conv`, and the initial value of the `cpt_nb_conv` loop counter is zero. Because of the abstraction and the interval computed for `Nb_Conv`, this abstract value computed by Astrée for `cpt_nb_conv` in the body of the loop is `[0, 341]`.

Then, using the range computed for `Index_1ere_Conv`, the abstract value for `Index_Conv` after instruction 7 is `[0, 1458]`.

In the concrete semantics of the program, `XMC_Ct_T_TABLE_GW_LIST[]` is a constant table of size 8192. It contains significant data up to index 1118, and all remaining locations have value $2^{32}$-1. Instruction 9 uses the `P_List_Index_Conv` pointer computed by instruction 8 to read the `GW_Name_Function` field of the element at index `Index_Conv` in this array.

From the abstract value computed for `Index_Conv` after instruction 7, i.e. `[0, 1458]`, Astrée considers that accesses to the `XMC_Ct_T_TABLE_GW_LIST[]` array beyond index 1118 are possible.

However, we have checked that no real execution of the program computes indexes greater than 1118, thus, the `Function_Id` variable cannot be assigned the $2^{32}$–1 value. Hence, this is a false alarm.

## 5.3 The Way to Avoid This False Alarm

Looking at the `XMC_Ct_T_TABLE_GW[]` array, we notice that all values of variables `Index_1ere_Conv` and `Nb_Conv` are bound by the following relation: `Index_1ere_Conv + Nb_Conv < 1118`. Such a relation is usually precisely caught by the octagon domain of Astrée. We have now to find out why the constraint computed in this case, i.e. `Index_1ere_Conv + Nb_Conv <= 1459`, is not precise enough.

This constraint is computed by Astrée after instruction 4. It is imprecise because the abstract value of `Nb_Conv` (resp. `Index_1ere_Conv`) results from the join of the values of the `Conv_Number` field (resp. `Begin_List_Index`) for all possible values of index `IdFctConv`, i.e. `[0, 51]`.

In order to force Astrée not to compute the above mentioned joins too early, we add partitioning directives into the code.

`__ASTREE_partition_begin((IdFctConv));` is inserted before instruction 2, while the related `__ASTREE_partition_merge(());` is inserted after instruction 4. These directives make Astrée perform a separate analysis for each individual possible value of `IdFctConv`.

The consequence is that the precise `Index_1ere_Conv + Nb_Conv <= 1119` constraint is now computed by Astrée after instruction 4.

Nevertheless, the alarm does not disappear. At this point, there is no way left for the user to tune the analysis better. That is a typical case in which support from the tool-developers is needed. After a slight improvement by the Astrée team dealing with product reduction between the interval and the octagon abstract domains, this alarm is no longer raised.

## 6 Conclusion

The experiments described in this paper show that the Astrée static analyser can be used by engineers from industry to prove the absence of RTE on real avionics programs, and that such non-expert users can meet the zero false alarms objective. Among the reasons for this success, one is to quote the fact that the user does not have to provide Astrée with the invariant of the program to be analysed, only a few clues on how to find it are necessary. The next step for this tool could be its transfer to operational software development teams, which requires an industrial version of Astrée, guaranteeing perennial support.

Moreover, our experience with tools like Astrée gives us the opportunity to sketch a customer-supplier relationship model that could be appropriate for abstract interpretation based tools.

Indeed, the specialisation process of a precise abstract interpretation based analyser makes it necessary for the tool designers to receive accurate information on the targeted type of programs from the end-users. The customer must therefore reveal detailed information about the structure of the targeted programs, their execution model, their dimensions and the type of computations they perform, and provide representative examples.

Furthermore, any change in the analysed program may cause the analyser to become too imprecise for the false alarm reduction process to be industrially feasible. If the case arises, the tool-supplier has to adapt the analyser. As a consequence, the providers of such tools must be prepared to update their products, e.g. add or improve abstract domains, whenever the set of parameters is no longer sufficient to analyse some program of the family precisely, even after the tool specialisation has been performed.

This kind of support comes on top of the usual list of services that any tool-provider has to offer.

In brief, a dedicated tool requires a one-to-one customer-supplier relationship.

*Acknowledgements.* We warmly thank every member of the Astrée team for their constant support.

## References

1. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proc. ACM SIGPLAN'2003 Conf. PLDI, San Diego, CA, US, June 7–14, 2003, pp. 196–207. ACM Press, New York (2003)
2. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREE analyser. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
3. Cousot, P.: Interprétation abstraite. Technique et Science Informatique, vol. 19, Nb 1-2-3. Janvier 2000, Hermès, Paris, France, pp. 155–164 (2000)
4. Cousot, P.: Abstract Interpretation Based Formal Methods and Future Challenges. In: Wilhelm, R. (ed.) Informatics. LNCS, vol. 2000, pp. 138–156. Springer, Heidelberg (2001)
5. Cousot, P., Cousot, R.: Basic Concepts of Abstract Interpretation. In: Jacquard, R. (ed.) Building the Information Society, pp. 359–366. Kluwer Academic Publishers, Dordrecht (2004)
6. Cousot, P., Cousot, R., Feret, J., Miné, A., Mauborgne, L., Monniaux, D., Rival, X.: Varieties of Static Analyzers: A Comparison with ASTREE (2007)
7. Souyris, J., Le Pavec, E., Himbert, G., Jégu, V., Borios, G., Heckmann, R.: Computing the worst-case execution time of an avionics program by abstract interpretation. In: Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis, pp. 21–24 (2005)
8. Goubault, E., Martel, M., Putot, S.: Static Analysis-Based Validation of Floating-Point Computations. In: Alt, R., Frommer, A., Kearfott, R.B., Luther, W. (eds.) Numerical Software with Result Verification. LNCS, vol. 2991, pp. 306–313. Springer, Heidelberg (2004)

# Magic-Sets Transformation for the Analysis of Java Bytecode

Étienne Payet[1] and Fausto Spoto[2]

[1] IREMIA, Université de la Réunion, France
[2] Dipartimento di Informatica, Università di Verona, Italy

**Abstract.** Denotational static analysis of Java bytecode has a nice and clean compositional definition and an efficient implementation with binary decision diagrams. But it models only the *functional i.e.,* input/output behaviour of a program $P$, not enough if one needs $P$'s *internal* behaviours *i.e.,* from the input to some internal program points. We overcome this limitation with a technique used up to now for logic programs only. It adds new *magic* blocks of code to $P$, whose functional behaviours are the internal behaviours of $P$. We prove this transformation correct with an operational semantics. We define an equivalent denotational semantics, whose denotations for the magic blocks are hence the internal behaviours of $P$. We implement our transformation and instantiate it with abstract domains modelling *sharing* of two variables and *non-cyclicity* of variables. We get a static analyser for full Java bytecode that is faster and scales better than another operational pair-sharing analyser and a constraint-based pointer analyser.

## 1 Introduction

Static analysis determines at compile-time properties about the run-time behaviour of computer programs. It is used for optimising their compilation [1], deriving loop invariants, verifying program annotations or security constraints [15]. This is very important for low-level languages such as Java bytecode, downloaded from insecure networks in a machine-independent, non-optimised format. Since its source code is not available, its direct analysis is desirable.

Correctness is usually mandatory for static analysis and proved *w.r.t.* a *reference semantics* of the analysed language. Abstract interpretation [8] shows here its strength since it derives static analyses *from the semantics* itself, so that they are by construction correct or even optimal. The derived analyses inherit semantical features such as compositionality and can only model program properties that can be formalised in terms of the reference semantics.

There are three main ways of giving semantics to a piece of code $c$ [21]: *operational semantics* models $c$'s execution as a transition relation over *configurations*, which include implementational details such as activation stacks and return points from calls; *denotational semantics* provides instead a *denotation i.e.,* a function from the input state provided to $c$ (the values of the variables before $c$ is executed) to the resulting output state (the same values after $c$ has been

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 452–467, 2007.

executed); *axiomatic semantics* derives the *weakest precondition* which must hold before the execution of $c$ from a given postcondition which holds after it.

```
public class List {
 private C head;  private List tail;

 public List(C head, List tail) {
  this.head = head;  this.tail = tail;
 }

 private List() {
  List cursor = null;
  for (int i = 5; i > 0; i--)
   cursor = new List(new C(i),cursor);
  head = new C(0);  tail = cursor;
 }

 public List clone() {
  if (tail == null) return new List(head,null);
  else return new List(head,tail.clone());
 }

 public List deepClone() {
  if (tail == null)
   return new List(head.clone(),null);
  else
   return new List(head.clone(),tail.deepClone());
 }

 public static void main(String[] args) {
  List v1 = new List();
  List v2 = v1.clone();
  v2 = v1.deepClone();
 }
}
```

**Fig. 1.** Our running example

A major drawback of denotational semantics is that denotations model only the *functional i.e.,* input/output behaviour of the code: they do not express its *internal i.e.,* input/internal program points behaviours. The derived static analyses inherit this drawback, which makes them often useless in practice, with some notable exceptions for analyses which are not focused on internal program points, such as strictness analysis. Consider the Java code in Fig. 1, which implements a list of `C`'s with two cloning methods: `clone` returns a shallow copy of a list and `deepClone` a deep copy, where also the `C`'s have been cloned. Hence, in `main`:

1. the return value of `clone` *shares* data structures with the list `v1`, namely, its `C`'s objects. Moreover, it is a *non-cyclical* list, since `v1` is a non-cyclical list;
2. the return value of `deepClone` does not share with `v1`, since it is a deep copy, and is also non-cyclical.

*Sharing* analysis of pairs of variables and *non-cyclicity* analysis of variables, based on denotational semantics and implemented with a *pair-sharing domain* [16] and a *non-cyclicity domain* [14], can only prove 2, since 1 needs information at the internal program point just after the call to `clone`. If we add a command at the end of `main`, they cannot even prove 2 anymore.

Years ago abstract interpretation was mainly applied to logic (sometimes functional) languages and denotational semantics was one of the standard reference semantics. The above problem about internal program points was solved with a *magic-sets transformation* of the program $P$, specific to logic languages, which adds extra *magic* clauses whose functional behaviours are the internal behaviours of $P$ [3,4,7]. Codish [6] kept the overhead of the transformation small by exploiting the large overlapping between the clauses of $P$ and the new magic clauses. Abstract interpretation has moved later towards mainstream imperative languages, even low-level ones such as Java bytecode. Suddenly, operational semantics became *the* reference semantics. This was a consequence of the lack of a magic-sets transformation for imperative languages and of the intuitive definition of operational semantics, very close to an actual implementation of the run-time engine of the language.

Our contributions here are the definition of a magic-sets transformation for Java bytecode, its proof of correctness, its implementation inside our Julia denotational analyser [17], its instantiation with two domains for pair-sharing [16] and non-cyclicity [14] and its evaluation and comparison with an operational analyser for pair-sharing [12] and the points-to analyser Spark [9]. Julia is one or two orders of magnitude faster. It scales to programs of up to 19000 methods, for which the other two analysers are not always applicable.

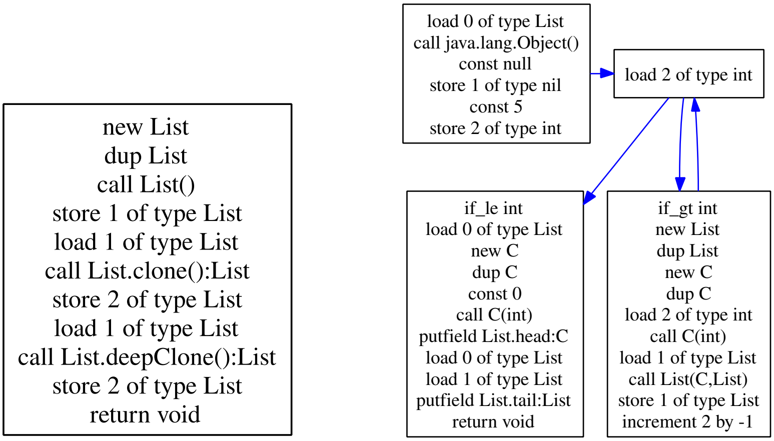
To understand why we want to *rediscover* denotational static analysis and why its implementation Julia is so efficient, consider the following:

– if method $m$ (constructor, function, procedure...) is called in program points $p_1, \ldots, p_n$, denotational analyses compute $m$'s denotation *only once* and then *extend* it at each $p_i$. Hence they can be very fast for analysing complex software where $n$ is often large. Operational analyses process instead $m$ from scratch *for every* $p_i$. *Memoisation*, which takes note of the input states for which $m$ has been already analysed and caches the results, is a partial solution to this problem, since each $p_i$ often calls $m$ with *different* input states;
– denotations *i.e.,* functions from input to output, can be represented as Boolean functions, namely, logical implications from the properties of the input to the properties of the output. Boolean functions have an efficient implementation as binary decision diagrams [5]. Hence there is a potentially very efficient implementation of denotational static analyses, which is not always the case for operational static analyses;
– denotational semantics is *compositional i.e.,* the denotation of a piece of code is computed bottom-up from those of its subcomponents (commands or expressions). The derived static analyses are hence compositional, an invaluable simplification when one formalises, implements and debugs them;
– denotational semantics does not use activation stacks nor return points from calls. Hence it is simpler to abstract than an operational semantics;
– denotational semantics models naturally properties of the functional behaviour of the code, such as information flows [15]. Operational semantics is very awkward here.

These are not *theoretical* insights, as our experiments show in Section 7.

## 2 Our Magic-Sets Transformation for Java Bytecode

The left of Fig. 2 reports the Java bytecode for the `main` method in Fig. 1, after a light preprocessing performed by our Julia analyser. It has a simple sequential control, being a single block of code. This is because we do not consider exceptions for simplicity, which are implicitly raised by some instructions and break the sequential structure of the code without changing the sense of our magic-sets



```
new List
dup List
call List()
store 1 of type List
load 1 of type List
call List.clone():List
store 2 of type List
load 1 of type List
call List.deepClone():List
store 2 of type List
return void
```

```
load 0 of type List
call java.lang.Object()
const null
store 1 of type nil
const 5
store 2 of type int
```

```
load 2 of type int
```

```
if_le int
load 0 of type List
new C
dup C
const 0
call C(int)
putfield List.head:C
load 0 of type List
load 1 of type List
putfield List.tail:List
return void
```

```
if_gt int
new List
dup List
new C
dup C
load 2 of type int
call C(int)
load 1 of type List
call List(C,List)
store 1 of type List
increment 2 by -1
```

**Fig. 2.** The Java bytecode of `main` and of the empty constructor of `List` in Fig. 1

transformation (our actual implementation in Julia considers exceptions). The code in Figure 2 is *typed i.e.,* instructions are decorated with the type of their operands, and *resolved i.e.,* method and field references are bound to their corresponding definition. For type inference and resolution we used the official algorithms [10]. A method or constructor implementation in class $\kappa$, named $m$, expecting parameters of types $\boldsymbol{\tau}$ and returning a value of type $t$ is written as $\kappa.m(\boldsymbol{\tau}) : t$. The `call` instruction implements the four `invoke`'s available in Java bytecode. It reports the explicit list of method or constructor implementations that it might call at run-time, accordingly with the semantics of the specific `invoke` that it implements. We allow more than one implementation for late-binding, but we use only one in our examples, for simplicity. Dynamic lookup of the correct implementation of a method is performed by *filter* instructions at the beginning of each method, which we do not show for simplicity.

Local variable 1 on the left of Fig. 2 implements variable `v1` in Fig. 1. Hence, just after the call to `clone`, it *shares* with the return value of `clone`, left on top of the stack, and is non-cyclical; after the call to `deepClone`, it *does not share* with the return value of `deepClone`, left on top of the stack, and is non-cyclical. To prove these results with a denotational analysis, our magic-sets transformation builds new *magic* blocks of code whose functional behaviours are the internal behaviours just after the calls to `clone` and `deepClone` on the left of Fig. 2.

Let us describe this transformation. It starts by splitting the code after the two calls to `clone` and `deepClone`, since we want to observe the intermediate states there. For reasons that will be clear soon, it also splits the code before each `call`. The result is in Fig. 3. The original code is split into blocks $0, \ldots, 5$ now. These have outgoing dashed arrows leading to new grey *magic* blocks $m0, \ldots, m5$. Block $mk$ contains the same bytecode as block $k$ plus a leading `blockcall` $mp$, where $p$ is the predecessor of block $k$, if any.

The functional behaviour of magic block $mk$ coincides with the internal behaviour at the end of block $k$. For instance, the functional behaviours of $m2$ and $m4$ are maps from the input state provided to the program to the intermediate states just after the calls to `clone` and `deepClone`, respectively. To understand why, let us start from $m0$. It is a clone of block 0 so that, at its end, the computation reaches the intermediate state at the internal program point between 0 and 1. Block $m1$ executes $m0$ (because of the `blockcall m0` instruction), then the same instructions as block 1. At its end, the computation reaches hence the intermediate state at the internal program point between 1 and 2. The same reasoning applies to the other magic blocks.

Consider the Java bytecode of the empty constructor of `List` in Fig. 1 now, called by `main` and shown on the right of Fig. 2. It is not sequential since it contains a loop. Its magic-sets transformation is in Fig. 4. As for `main`, we split the original code before each `call`; each magic block $mk$ contains the code of $k$ plus a leading `blockcall` to the predecessor(s) of $k$, if any. Since 8 has two predecessors 7 and 11, block $m8$ starts with `blockcall m7 m11` *i.e.,* there are two ways of reaching 8 and the states observable at its end are obtained by executing `load 2 of type int` from a state reachable at the end of 7 or 11. Something new happens for $m6$. It starts with a call to $m0$ in Fig. 3, which provides the intermediate states just before the only call in the program to this empty constructor of `List`. Block $m6$ continues with a `makescope List()` instruction which builds the scope of the constructor: in Java bytecode the caller stores the actual arguments on the operand stack and the callee retrieves them from the local variables [10]. Hence `makescope List()` copies the `List` object, left on top of the stack by $m0$ (*i.e.,* the implicit `this` parameter), into local variable 0 and clears the stack. At the end of $m6$ we observe hence the states reachable at the internal program point between blocks 6 and 7. This is why

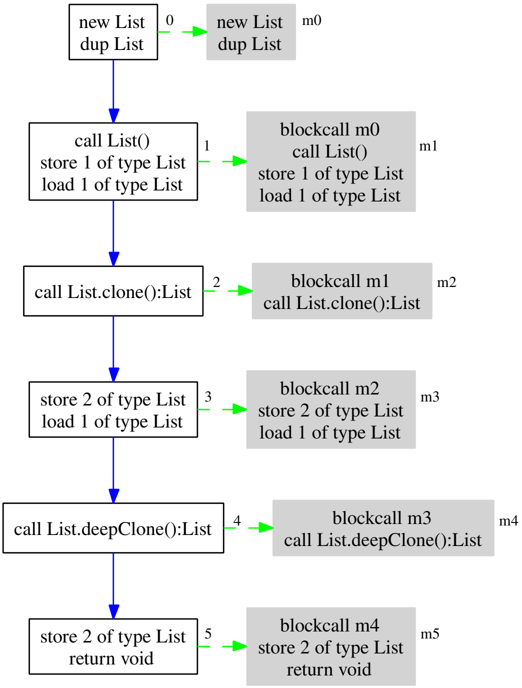


**Fig. 3.** The magic-sets transformation of the Java bytecode on the left of Fig. 2

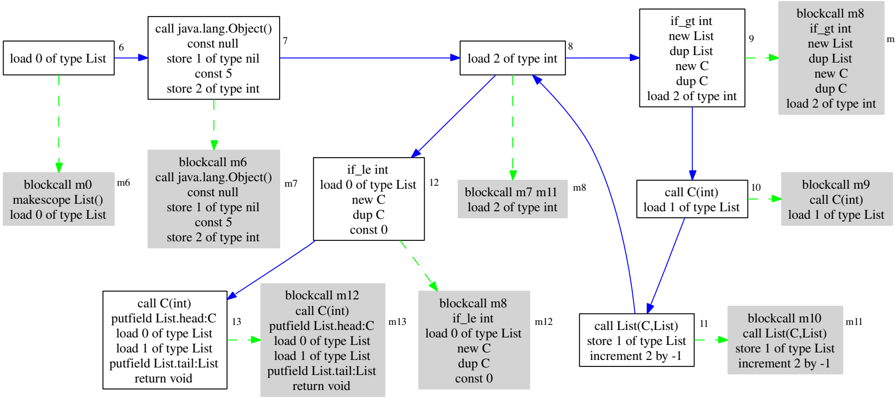


**Fig. 4.** The magic-sets transformation of the constructor on the right of Fig. 2

we split the code before each `call`: to allow the states of the callers at the call points to flow into the callees.

## 3 A Formalisation of Our Magic-Sets Transformation

We formalise here the magic-sets transformation. From now on we assume that $P$ is a program *i.e.,* a set of blocks as those in Fig. 3 and Fig. 4. We assume that the starting block of a method has no predecessors and does not start with a `call`, without loss of generality since it is always possible to add an extra initial block containing `nop`; we assume that the other blocks have at least a predecessor, since otherwise they would be dead-code and eliminated; we assume that each `call` starts a block and that each `return` ends a block with no successors; we assume that the `main` method is not called from inside the program, without loss of generality since we can always rename `main` into `main'` wherever in the program and add a new `main` which wraps a call to `main'`.

Original blocks are labelled with $k$ and magic blocks with $mk$ with $k \in \mathbb{N}$. If $\ell$ is a label, $P(\ell)$ is block $\ell$ of $P$. We write block $\ell$ with $n$ bytecode instructions and $m$ immediate successor blocks $b_1, \ldots, b_m$, with $m, n \geq 0$, as

$$\boxed{\begin{array}{c}\texttt{ins}_1\\ \texttt{ins}_2\\ \cdots\\ \texttt{ins}_n\end{array}}^{\ell} \rightrightarrows \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \qquad \text{or just as} \qquad \boxed{\begin{array}{c}\texttt{ins}_1\\ \texttt{ins}_2\\ \cdots\\ \texttt{ins}_n\end{array}}^{\ell} \qquad \text{when } m = 0.$$

The *magic-sets transformation* of $P$ builds a *magic block* $mk$ for each block $k$.

**Definition 1.** *The* magic block *mk, with $k \in \mathbb{N}$, is built from $P(k)$ as*

$$magic\Bigg(\underbrace{\boxed{code}\overset{k}{\rightrightarrows}\begin{smallmatrix}b_1\\ \cdots\\ b_m\end{smallmatrix}}_{P(k)}\Bigg) = \begin{cases} \boxed{\begin{array}{c}\texttt{blockcall}\ mp_1\cdots mp_l\\ code\end{array}}^{mk} & \textit{if } l>0 \\ \boxed{\begin{array}{c}\texttt{blockcall}\ mq_1\cdots mq_u\\ \texttt{makescope}\ \kappa.m(\boldsymbol{\tau}){:}t\\ code\end{array}}^{mk} & \textit{if } l=0 \textit{ and } u>0 \\ \boxed{code}^{mk} & \textit{if } l=0 \textit{ and } u=0 \end{cases}$$

*where $p_1,\ldots,p_l$ are the predecessors of $P(k)$ and $q_1,\ldots,q_u$ those of the blocks of $P$ which begin with a* `call` *to the method $\kappa.m(\boldsymbol{\tau}) : t$ starting at block $k$.* □

Definition 1 has three cases. In the first case block $k$ does not start a method (or constructor). Hence it has $l > 0$ predecessors and magic block *mk* begins with a `blockcall` to their magic blocks, as block *m8* in Fig. 4. In the second and third case block $k$ starts a method or constructor $\kappa.m(\boldsymbol{\tau}) : t$, so that it has no predecessors. If the program $P$ calls $\kappa.m(\boldsymbol{\tau}) : t$ (second case) there are $u > 0$ predecessors of those `call`s, since we assume that `call` does not start a method. Magic block *mk* calls those predecessors and then uses `makescope` to build the scope for $\kappa.m(\boldsymbol{\tau}) : t$, as for block *m6* in Fig. 4. Otherwise (third case), $P$ never calls $\kappa.m(\boldsymbol{\tau}) : t$ and *mk* is a clone of $k$, as for block *m0* in Fig. 3.

## 4 Operational Semantics of the Java Bytecode

In this section we describe an operational semantics of the Java bytecode, which we use in Section 5 to prove our magic-sets transformation correct.

**Definition 2.** *A* state *of the Java Virtual Machine is a triple $\langle l \,\|\, s \,\|\, \mu\rangle$ where $l$ maps local variables to* values*, $s$ is a stack of values (the* operand stack*), which grows leftwards, and $\mu$ is a* memory*, or* heap*, which maps* locations *into* objects*. We do not formalise further what values, memories and objects are, since this is irrelevant here. The set of states is $\Sigma$.* □

The semantics of a bytecode instruction `ins` different from `call` and `blockcall` is a partial map *ins* from states to states. For instance, the semantics of `dup` $t$ is

$$dup\ \ t = \lambda\langle l \,\|\, s \,\|\, \mu\rangle.\langle l \,\|\, top :: s \,\|\, \mu\rangle$$

where $s = top :: s'$ and $top$ has type $t$. This is always true since legal Java bytecode is verifiable [10]. The semantics of `load` $i$ `of type` $t$ is

$$load\ \ i\ \ of\ \ type\ \ t = \lambda\langle l \,\|\, s \,\|\, \mu\rangle.\langle l \,\|\, l(i) :: s \,\|\, \mu\rangle$$

where $l(i)$ exists and has type $t$ since legal Java bytecode is verifiable.

Also the semantics of a `return` $t$ bytecode is a map over states, which leaves on the operand stack only those elements which hold the return value of type $t$:

$$return\ \ t = \lambda\langle l \,\|\, s \,\|\, \mu\rangle.\langle l \,\|\, vs \,\|\, \mu\rangle$$

where $s = vs :: s'$ and $vs$ are the stack elements which hold the return value. If $t = \texttt{void}$ then $vs = \varepsilon$. We formalise later in Definition 5 how control returns to the caller. Also the semantics of a conditional bytecode is a map over states, undefined when its condition is false. For instance, the semantics of `if_le` $t$ is

$$\mathit{if\_le}\ t = \lambda\langle l \parallel s \parallel \mu\rangle.\begin{cases}\langle l \parallel s' \parallel \mu\rangle & \text{if } \mathit{top} \le 0\\ \mathit{undefined} & \text{otherwise,}\end{cases}$$

where $s = \mathit{top} :: s'$ and $\mathit{top}$ has numerical type $t$.

When a caller transfers the control to a callee $\kappa.m(\boldsymbol{\tau}) : t$, the Java Virtual Machine performs an operation *makescope* $\kappa.m(\boldsymbol{\tau}) : t$ which copies the topmost stack elements into the corresponding local variables and clears the stack.

**Definition 3.** *Let $\kappa.m(\boldsymbol{\tau}) : t$ be a method or constructor and $p$ the number of stack elements needed to hold its actual parameters, including the implicit parameter* `this`*, if any. We define $(\mathit{makescope}\ \kappa.m(\boldsymbol{\tau}) : t) : \Sigma \to \Sigma$ as*

$$\mathit{makescope}\ \kappa.m(\boldsymbol{\tau}) : t = \lambda\langle l \parallel s \parallel \mu\rangle.\langle[i \mapsto v_i \mid 0 \le i < p] \parallel \varepsilon \parallel \mu\rangle,$$

*where $s = v_{p-1} :: \cdots :: v_0 :: s'$ since legal Java bytecode is verifiable.* □

Definition 3 formalises the fact that the $i$th local variable of the callee is a copy of the element $p-1-i$ positions down the top of the stack of the caller.

**Definition 4.** *A* configuration *is a pair $\langle b \parallel \sigma\rangle$ of a block $b$ (not necessarily in $P$) and a state $\sigma$. It represents the fact that the Java Virtual Machine is going to execute $b$ in state $\sigma$. An* activation stack *is a stack $c_1 :: c_2 :: \cdots :: c_n$ of configurations, where $c_1$ is the topmost,* current *or* active *configuration.* □

We can define now the *operational semantics* of a Java bytecode program.

**Definition 5.** *The (small step) operational semantics of a Java bytecode program $P$ is a relation $a' \Rightarrow_P a''$ ($P$ is usually omitted) providing the immediate successor activation stack $a''$ of an activation stack $a'$. It is defined by the rules:*

$$\frac{\texttt{ins}\ \textit{is not a}\ \texttt{call}\ \textit{nor a}\ \texttt{blockcall}}{\langle \boxed{\begin{array}{c}\texttt{ins}\\ \mathit{rest}\end{array}} \overset{\ell}{\rightrightarrows} \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \parallel \sigma\rangle :: a \Rightarrow \langle \boxed{\mathit{rest}} \overset{\ell}{\rightrightarrows} \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \parallel \mathit{ins}(\sigma)\rangle :: a} \tag{1}$$

$$\frac{\begin{array}{c}b \textit{ is the block where method } \kappa.m(\boldsymbol{\tau}) : t \textit{ starts}\\ \sigma = \langle l \parallel \mathit{pars} :: s \parallel \mu\rangle, \quad \mathit{pars} \textit{ are the actual parameters of the call}\\ \sigma' = (\mathit{makescope}\ \kappa.m(\boldsymbol{\tau}) : t)(\sigma)\end{array}}{\langle \boxed{\begin{array}{c}\texttt{call}\ \kappa.m(\boldsymbol{\tau}){:}t\\ \mathit{rest}\end{array}} \overset{\ell}{\rightrightarrows} \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \parallel \sigma\rangle :: a \Rightarrow \langle b \parallel \sigma'\rangle :: \langle \boxed{\mathit{rest}} \overset{\ell}{\rightrightarrows} \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \parallel \langle l \parallel s \parallel \mu\rangle\rangle :: a} \tag{2}$$

$$\frac{}{\langle \boxed{\phantom{x}}^{k} \parallel \langle l \parallel vs \parallel \mu\rangle\rangle :: \langle b \parallel \langle l' \parallel s' \parallel \mu'\rangle\rangle :: a \Rightarrow \langle\, b \parallel \langle l' \parallel vs :: s' \parallel \mu\rangle\rangle :: a} \tag{3}$$

$$\frac{1 \le i \le m}{\langle \boxed{\phantom{x}}^{k} \rightrightarrows \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \parallel \sigma\rangle :: a \Rightarrow \langle b_i \parallel \sigma\rangle :: a} \tag{4}$$

$$\frac{1 \leq i \leq l}{\langle \boxed{\begin{array}{c}\texttt{blockcall}\ mp_1 \cdots mp_l\\ rest\end{array}}^{mk} \parallel \sigma\rangle :: a \Rightarrow \langle P(mp_i) \parallel \sigma\rangle :: \langle \boxed{rest}^{mk} \parallel \sigma\rangle :: a} \quad (5)$$

$$\frac{}{\langle \boxed{\phantom{x}}^{mk} \parallel \sigma\rangle :: \langle b \parallel \sigma'\rangle :: a \Rightarrow \langle\, b \parallel \sigma\rangle :: a} \quad (6)$$

□

Rule (1) executes an instruction `ins`, different from `call` and `blockcall`, by using its semantics *ins*. The Java Virtual Machine moves then forward to run the rest of the instructions. Instruction `ins` might be here a `makescope`, whose semantics is given in Definition 3. Rule (2) calls a method. It looks for the block $b$ where the latter starts and builds its initial state $\sigma'$, by using *makescope*. It creates a new current configuration containing $b$ and $\sigma'$. It removes the actual arguments from the old current configuration and the call from the instructions still to be executed at return time. Control returns to the caller by rule (3), which rehabilitates the configuration of the caller but forces the memory to be that at the end of the execution of the callee. The return value of the callee is pushed on the stack of the caller. Rule (4) applies when all instructions inside a block have been executed; it runs one of its immediate successors, if any. This rule is normally deterministic, since if a block of the Java bytecode has two or more immediate successors then they start with mutually exclusive conditional instructions and only one thread of control is actually followed. Rule (5) runs a `blockcall` by choosing one of the called blocks $mp_i$ and creating a new configuration where it can run. This is true non-determinism, corresponding to the fact that there might be more ways of reaching a magic block and hence more intermediate states at an internal program point. Rule (6) applies at the end of the execution of a magic block *mk*. It returns the control to the caller of *mk* and keeps the state reached at the end of the execution of *mk*. Rules (1) and (2) can be used both for the original and for the magic blocks of the program; rules (3) and (4) only for the original blocks; rules (5) and (6) only for the magic ones.

Our small step operational semantics allows us to define the set of intermediate states at a given, internal program point $*$, provided $*$ ends a block. This can always be obtained by splitting after $*$ the block where $*$ occurs.

**Definition 6.** *Let $\sigma_{in}$ be the initial state provided to the method* `main` *of $P$ starting at block $b_{in}$. The* intermediate states *at the end of block $k \in \mathbb{N}$ during the execution of $P$ from $\sigma_{in}$ are*

$$\Sigma_k = \{\sigma \mid \langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{\phantom{xx}}^{k} \rightrightarrows \begin{smallmatrix} b_1\\ \cdots\\ b_m\end{smallmatrix} \parallel \sigma\rangle :: a\}$$

□

Note that $\Sigma_k$ is in general a set since there might be more ways of reaching block $k$, for instance through loops or recursion.

## 5 Correctness of the Magic-Sets Transformation

By using the operational semantics of Section 4, we show that the final states reached at the end of the execution of a magic block $mk$ are exactly the intermediate states reached at the end of block $k$, before executing its successors: the functionalbehaviour of $mk$ coincides with the internalbehaviour at the end of $k$.

**Theorem 1.** *Let $\sigma_{in}$ be the initial state provided to the* `main` *method of $P$ and $k \in \mathbb{N}$ a block of $P$. We have $\Sigma_k = \left\{\sigma \,\middle|\, \langle P(mk) \,\|\, \sigma_{in}\rangle \Rightarrow^* \langle \boxed{\phantom{xx}}^{mk} \,\|\, \sigma\rangle \right\}$.* □

In Section 6 we define a denotational semantics for the Java bytecode and prove it *equivalent* to our operational semantics of Section 4 *w.r.t.* functional behaviours. By Theorem 1, we will conclude that the denotational semantics of $mk$ is the internal behaviour at the end of block $k$.

## 6 Denotational Semantics of the Java Bytecode

A denotational semantics for the Java bytecode maps each block of code $b$ in a *denotation* $[\![b]\!]$ *i.e.,* in a partial function from an *initial* state at the beginning of $b$ to an *output* or *final* state at the end of the execution of the code starting at $b$. Hence, if $b_{in}$ is the initial block of method `main`, then $[\![b_{in}]\!]$ is the functional behaviour of the whole program.

**Definition 7.** *A denotation is a partial function from an* input *state to an* output *or* final *state. The set of denotations is written as $\Delta$. Let $\delta_1, \delta_2 \in \Delta$. Their* sequential composition *is $\delta_1;\delta_2 = \lambda\sigma.\delta_2(\delta_1(\sigma))$, which is undefined when $\delta_1(\sigma)$ is undefined or when $\delta_2(\delta_1(\sigma))$ is undefined.* □

It follows that the semantics *ins* of a bytecode `ins` is a denotation.

Let $\delta \in \Delta$ be the functional behaviour of a method $\kappa.m(\boldsymbol{\tau}) : t$. At its beginning the operand stack is empty and the local variables hold the actual arguments of the call. At its end the operand stack holds the return value of $\kappa.m(\boldsymbol{\tau}) : t$ only, if any (the semantics of `return` drops all stack elements but the return value. See Section 4). From the point of view of a caller executing a `call` $\kappa.m(\boldsymbol{\tau}) : t$, the local variables and the operand stack do not change, except for the actual arguments which get popped from the stack and substituted with the return value, if any. The final memory is that reached at the end of $\kappa.m(\boldsymbol{\tau}) : t$. These considerations let us *extend* $\delta$ into the denotation of the `call` instruction.

**Definition 8.** *Let $\delta \in \Delta$ and $\kappa.m(\boldsymbol{\tau}) : t$ be a method. We define the operator $extend\ \kappa.m(\boldsymbol{\tau}) : t \in \Delta \mapsto \Delta$ as*

$$(extend\ \kappa.m(\boldsymbol{\tau}) : t)(\delta) = \lambda\langle l \,\|\, pars :: s \,\|\, \mu\rangle.\langle l \,\|\, vs :: s \,\|\, \mu'\rangle$$

*where $\langle l' \,\|\, vs \,\|\, \mu'\rangle = \delta((makescope\ \kappa.m(\boldsymbol{\tau}) : t)(\langle l \,\|\, pars :: s \,\|\, \mu\rangle))$, $pars$ are the actual parameters passed to $\kappa.m(\boldsymbol{\tau}) : t$ and $vs$ its return value, if any.* □

An *interpretation* is a set of denotations for each block of $P$. *Sets* can express non-deterministic behaviours, which means for us that we can observe more intermediate states between blocks. The operations *extend* and ; over denotations are consequently extended to sets of denotations.

**Definition 9.** *An* interpretation *for $P$ is a map from $P$'s blocks into sets of denotations. The set of interpretations $I$ is ordered by pointwise set-inclusion.* □

Given an interpretation $\iota$ providing the functional behaviour of the blocks of $P$, we can determine the functional behaviour $[\![b]\!]^{\iota}$ of the code starting at a given block $b$, not necessarily in $P$, which can call methods and blocks of $P$.

**Definition 10.** *Let $\iota \in I$. The* denotations in $\iota$ of an instruction *are*

$$\begin{aligned}
[\![\texttt{ins}]\!]^{\iota} &= \{ins\} \quad \textit{if } \texttt{ins} \textit{ is not a } \texttt{call} \textit{ nor a } \texttt{blockcall}\\
[\![\texttt{blockcall}\ mp_1 \cdots mp_l]\!]^{\iota} &= \iota(P(mp_1)) \cup \cdots \cup \iota(P(mp_l))\\
[\![\texttt{call}\ \kappa.m(\boldsymbol{\tau}) : t]\!]^{\iota} &= (\mathit{extend}\ \kappa.m(\boldsymbol{\tau}) : t)(\iota(b_{\kappa.m(\boldsymbol{\tau}):t}))
\end{aligned}$$

*where $b_{\kappa.m(\boldsymbol{\tau}):t}$ is the block where method or constructor $\kappa.m(\boldsymbol{\tau}) : t$ starts. The function $[\![\_]\!]^{\iota}$ is extended to blocks as*

$$\left[\!\!\left[ \boxed{\begin{array}{c}\texttt{ins}_1\\ \cdots\\ \texttt{ins}_\texttt{n}\end{array}}^{\ell} \rightrightarrows \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \right]\!\!\right]^{\iota} = \begin{cases} [\![\texttt{ins}_1]\!]^{\iota}; \cdots ; [\![\texttt{ins}_\texttt{n}]\!]^{\iota} & \textit{if } m = 0\\ [\![\texttt{ins}_1]\!]^{\iota}; \cdots ; [\![\texttt{ins}_\texttt{n}]\!]^{\iota}; (\iota(b_1) \cup \cdots \cup \iota(b_m)) & \textit{if } m > 0. \end{cases}$$

*with the assumption that if $n = 0$ then $[\![\texttt{ins}_1]\!]^{\iota}; \cdots ; [\![\texttt{ins}_\texttt{n}]\!]^{\iota} = \{id\}$, where the identity denotation $id$ is such that $id = \lambda\sigma.\sigma$.* □

The blocks of $P$ are in general interdependent, because of loops and method calls, and a denotational semantics must be built through a fixpoint computation. Given an empty approximation $\iota \in I$ of the denotational semantics, one improves it into $T_P(\iota) \in I$ and iterates the application of $T_P$ until a fixpoint[1].

**Definition 11.** *The* transformer $T_P : I \mapsto I$ *for $P$ is defined as $T_P(\iota)(b) = [\![b]\!]^{\iota}$ for every $\iota \in I$ and block $b$ of $P$.* □

**Proposition 1.** *The operator $T_P$ is additive, so its least fixpoint exists [20].* □

**Definition 12.** *Let $P$ be a Java bytecode program (possibly enriched with its magic blocks). Its* denotational semantics $\mathcal{D}_P$ *is the least fixpoint $\sqcup_{i \geq 0} T_P^i$ of $T_P$, where $T_P^0(b) = \varnothing$ for every block $b$ of $P$ and $T_P^{i+1} = T_P(T_P^i)$ for every $i \geq 0$.* □

We show now that the operational semantics of Section 4 and the denotational semantics of this section coincide, so that (Theorem 1) the denotation of a magic block $mk$ is the internal behaviour at the end of block $k$.

**Theorem 2.** *Let $b$ a block (not necessarily of $P$) and $\sigma_{in}$ an initial state for $b$. The functional behaviour of $b$, as modelled by the operational semantics of Section 4, coincides with its denotational semantics:*

$$\{\sigma_{out} \mid \langle b \,\|\, \sigma_{in} \rangle \Rightarrow_P^* \langle b' \,\|\, \sigma_{out} \rangle \not\Rightarrow_P\} = \{\delta(\sigma_{in}) \mid \delta \in [\![b]\!]^{\mathcal{D}_P},\ \delta(\sigma_{in}) \textit{ is defined }\}. \ \square$$

[1] Our implementation Julia performs smaller fixpoints on each strongly-connected component of blocks rather than a huge fixpoint over all blocks. This is important for efficiency reasons but irrelevant here for our theoretical results.

## 7 Experiments

We have implemented our magic-sets transformation inside the generic analyser JULIA for Java bytecode [17] and used it with two abstract domains. The first [16] overapproximates the set of pairs of program variables, which for the Java bytecode means local variables or stack elements, which *share i.e.,* reach the same memory location; it is used for automatic program parallelisation and to support other analyses. The second [14] overapproximates the set of *cyclical* program variables, those which reach a loop of memory locations; it needs a preliminary pair-sharing analysis. We used Boolean formulas to abstract sets of denotations by relating properties of their input to properties of their output. For instance, $(l1, s1) \Rightarrow (l1, l2)$ abstracts those denotations $\delta$ such that for every state $\sigma$, where only local variable 1 and stack element 1 might share (the base of the stack is $s0$), we have that in $\delta(\sigma)$ only local variables 1 and 2 might share (for simplicity, we do not report variables sharing with themselves [16]). We have implemented Boolean formulas through binary decision diagrams [5].

Let us consider pair-sharing. JULIA computes the formula $(l1, s0)$ as abstract denotation for block $m2$ in Fig. 3. It states that $(l1, s0)$ is true for $m2$ *i.e.,* at its end, only local variable 1, which holds the list `v1` of Fig. 1, might share with stack element 0 (the base of the stack), which holds the return value of `clone`. Hence JULIA proves that all other pairs of local variables and stack elements definitely do not share. This is an optimal approximation of the behaviour of the program between blocks 2 and 3. JULIA computes $(l1, l2)$ as abstract denotation for block $m4$. Hence it proves that local variables $l1$ (`v1` in Fig. 1) and $l2$ (`v2` in Fig. 1) *might* share there, while all other local variables and stack elements definitely do not share; in particular, the return value of `deepClone` (the stack element 0) does not share with `v1`. Note that `v1` and `v2` actually share after the call to `deepClone`, whose return value has not been stored into `v2` yet. This is an optimal approximation of the behaviour of the program between blocks 4 and 5. Let us consider cyclicity analysis. JULIA computes *false* as abstract denotation of both $m2$ and $m4$ in Fig. 3 *i.e.,* it proves that no local variable and stack element might be cyclical there, which is an optimal approximation of the behaviour of the program between blocks 2 and 3 and between blocks 4 and 5. In conclusion, JULIA proves both points 1 and 2 of Section 1.

Fig. 1 shows a simple program. More complex benchmarks such as those in Fig. 5 challenge the scalability, the efficiency and the precision of the analyses. The first 4 benchmarks, which are the smallest, have been also analysed with the pair-sharing analyser in [12] so we can build a comparison. The others are progressively larger to check the scalability of the analyses. Fig. 5 reports their size (number of methods), their preprocessing time with JULIA (extraction and parsing of the `.class` files, building a high-level representation of the bytecode and the magic-sets) and its percentage due to the magic-sets transformation, which is never more than 31%. We consider two scenarios: whether the Java libraries are not analysed (calls to the missing classes use a worst-case assumption) or they are analysed, for more precise but more costly analyses. We used

| | libraries are not included | | | libraries are included | | |
|---|---|---|---|---|---|---|
| | methods | preproc. | magic-sets | methods | preproc. | magic-sets |
| Qsort | 8 | 369 | 2.14% | 72 | 767 | 3.65% |
| IntegerQsort | 9 | 369 | 2.14% | 72 | 765 | 3.66% |
| Passau | 10 | 351 | 1.51% | 13 | 388 | 1.19% |
| ZipVector | 13 | 395 | 2.48% | 76 | 778 | 4.08% |
| JLex | 130 | 1292 | 14.53% | 744 | 2160 | 10.97% |
| JavaCup | 293 | 1502 | 9.8% | 1136 | 2657 | 21.88% |
| julia | 1441 | 3351 | 13.56% | 4809 | 8552 | 14.9% |
| jess | 1506 | 3344 | 25.1% | 6046 | 9911 | 28.88% |
| jEdit | 2473 | 6887 | 22.74% | 7943 | 15156 | 30.77% |
| soot | 15617 | 75925 | 10.49% | 19032 | 84709 | 14.54% |

**Fig. 5.** Size and preprocessing times (in milliseconds) for our benchmarks

| | **sharing analysis** | | | | **cyclicity analysis** | | | |
|---|---|---|---|---|---|---|---|---|
| | libr. not included | | libr. included | | libr. not included | | libr. included | |
| | time | precision | time | precision | time | precision | time | precision |
| Qsort | 127 | 35.09% | 267 | 71.79% | 20 | 0.00% | 43 | 10.71% |
| IntegerQsort | 208 | 36.17% | 295 | 53.46% | 23 | 0.00% | 38 | 17.18% |
| Passau | 152 | 36.88% | 118 | 43.03% | 14 | 5.88% | 6 | 100.00% |
| ZipVector | 251 | 21.15% | 395 | 40.47% | 34 | 0.00% | 50 | 7.69% |
| JLex | 1438 | 30.34% | 2312 | 33.86% | 269 | 17.00% | 877 | 32.64% |
| JavaCup | 2418 | 16.26% | 4996 | 22.96% | 474 | 14.78% | 1836 | 29.04% |
| julia | 10829 | 11.03% | 33589 | 12.22% | 2852 | 9.95% | 5245 | 17.23% |
| jess | 24526 | 12.79% | 66163 | 15.96% | 4136 | 8.19% | 8293 | 24.05% |
| jEdit | 42332 | 16.34% | 135208 | 19.92% | 6654 | 4.95% | 15926 | 10.33% |
| soot | 125819 | 6.26% | 282923 | 7.69% | 113884 | 9.89% | 196456 | 11.82% |

**Fig. 6.** Time (in milliseconds) and precision of our sharing and cyclicity analyses

an Intel Xeon machine running at 2.8GHz, with 2.5 gigabytes of RAM, Linux 2.6.15 and Sun jdk 1.5.

Fig. 6 reports the results of pair-sharing and cyclicity analyses with JULIA. Precision, for sharing analysis, is the percentage of pairs of distinct local variables or stack elements which are proved not to share, definitely, before a `putfield`, an `arraystore` or a `call`. We consider only variables and stack elements of reference type since primitive types cannot share in Java (bytecode); only `putfield`'s, `arraystore`'s and `call`'s since it is there that sharing analysis helps other analyses (see for instance [14] for its help to cyclicity analysis). Precision, for cyclicity analysis, is the percentage of local variables or stack elements which are proved to be non-cyclical, definitely, before a `getfield` bytecode. We consider only variables and stack elements of reference type since primitive values are never cyclical; only

| | Julia | [12] |
|---|---|---|
| Qsort | 496 | ≥1625 |
| IntegerQsort | 577 | ≥1335 |
| Passau | 502 | ≥1595 |
| ZipVector | 646 | ≥2780 |

| | Julia | [9] | | Julia | [9] |
|---|---|---|---|---|---|
| Qsort | 1.0 | 93 | JavaCup | 7.7 | 99 |
| IntegerQsort | 1.1 | 92 | julia | 42.1 | fails |
| Passau | 0.5 | 89 | jess | 76.1 | fails |
| ZipVector | 1.2 | 90 | jEdit | 150.4 | fails |
| JLex | 4.5 | 95 | soot | 367.7 | 452 |

**Fig. 7.** The table on the left reports the times (in milliseconds) of our pair-sharing analysis and of that in [12]. The table on the right reports the times (in seconds) of our pair-sharing analysis and of the points-to analysis in [9].

`getfield`'s since cyclicity information is typically used there, for instance to prove termination of iterations over dynamic data-structures [18]. For better efficiency, we *cache* the analysis of each bytecode so that, if it is needed twice, we only compute it once. This happens frequently with our magic-sets transformation, which introduces code duplication. For instance, block *m1* in Figure 3 shares three bytecodes which block 1. This technique has been inspired by a similar optimisation of the analysis of *magic logic programs*, defined in [6]. Since it caches the *functional* behaviour of the code, it is different from *memoisation*, which only caches its behaviour for *each given* input state.

We are not aware of any other cyclicity analysis for Java bytecode. An operational pair-sharing analyser was instead applied [12] to the smallest 4 benchmarks in Fig. 5, without including the library classes, but we could not use their analyser. It takes time $P + T + A$: $P$ is the preprocessing time, which they do not report. Since they use the generic analyser Soot, we could compute $P$ with Soot version 2.2.2; $T$ is the time to transform the output of Soot into the format required in [12]. We cannot estimate $T$ without the analyser; $A$ is the *preliminary running time* reported in [12], normalised *w.r.t.* the relative speeds of our machine and theirs. Fig. 7 on the left compares the running time of Julia, including preprocessing and without analysing the libraries, with $P + A$, since $T$ is unknown. Julia is faster, even without $T$. Exceptions, subroutines, static initialisers and native methods are not tested by such small benchmarks, so it is not clear if the analyser in [12] is ready for *real* analyses. Precision is expressed as a *level of multivariance* which we cannot translate into our more natural notion. Another analysis for (definite) sharing is implemented in [13] for a *subset* of Java. Times and precision are not reported. The code is not publicly available.

We compared our pair-sharing analysis with the Spark [9] points-to analysis, also based on Soot. Points-to and sharing information are somehow similar. We compare Spark against Julia including all Java libraries, since that is what Spark does. Soot, Spark and Julia are all written in Java. Hence a comparison is relatively fair. Fig. 7 on the right compares the overall running times, including preprocessing. Julia is always faster, up to two orders of magnitude as in the case of Passau; it completes all analyses while Spark stops in three cases with hard-to-understand run-time errors (for jEdit and jess: *This operation requires resolving level hierarchy but* `someclass` *is at resolving level dangling*; for julia: *couldn't find class* `jdd.bdd.BDD`, which does not exist and is not used by Julia). We stress that sharing and points-to analyses are anyway different analyses and neither of them is an abstraction of the other. Hence this comparison only indicates that Julia compares well *w.r.t.* an existing tool.

The Cibai tool [11] is able to derive class invariants from Java *source code* rather than from bytecode. It currently includes an abstract domain tracking *abstract locations*, which should provide some sharing information, although this is not detailed in [11]. No precision about sharing analysis is reported by the tool. It has been applied to programs of a few hundreds methods, which is still away from the 19032 methods of Julia (Figure 5). The tool is not freely available on the net so we could not build a comparison.

## 8 Conclusion

Our experiments show that denotational analyses of Java bytecode, with a preliminary magic-sets transformation, are feasible, fast and compare well with other analyses. We will soon use widenings [8] to further improve their efficiency.

Our magic-sets transformation is completely independent from the abstract domains, which can be developed without even knowing its existence. Then all abstract domains defined so far for the analysis of Java bytecode can in principle be used in our framework. The domain developer must only specify the internal program points where he wants to observe the results of the analysis, which depends on the specific goal for which he develops the abstract domain.

The efficiency of the static analyses based on our magic-sets transformation is partially a consequence of the fact that it enables the use of binary decision diagrams to represent the abstract domain elements. Then a challenge of our approach is its application to abstract domains which are difficult to represent through binary decision diagrams, such as those for class analysis [19] and path-length analysis [18]. In the first case we are using set-constraints; in the second case we are using the Parma Polyhedra Library [2]. The results of these experiments should confirm the applicability of our framework.

## References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles Techniques and Tools. Addison Wesley Publishing Company, Reading (1986)
2. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy (2006), `www.cs.unipr.it/Publications`
3. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.: Magic Sets and Other Strange Ways to Implement Logic Programs. In: Proc. of the 5th ACM Symposium on Principles of Database Systems, pp. 1–15. ACM Press, New York (1986)

4. Beeri, C., Ramakrishnan, R.: On the Power of Magic. The Journal of Logic Programming 10(3 & 4), 255–300 (1991)
5. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers 35(8), 677–691 (1986)
6. Codish, M.: Efficient Goal Directed Bottom-Up Evaluation of Logic Programs. Journal of Logic Programming 38(3), 355–370 (1999)
7. Codish, M., Dams, D., Yardeni, E.: Bottom-up Abstract Interpretation of Logic Programs. Journal of Theoretical Computer Science 124, 93–125 (1994)
8. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77), pp. 238–252. ACM Press, New York (1977)
9. Lhoták, H., Hendren, L.: Scaling Java Points-to Analysis using Spark. In: Hedin, G. (ed.) CC 2003 and ETAPS 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)
10. Lindholm, T., Yellin, F.: The Java$^{TM}$ Virtual Machine Specification, 2nd edn. Addison-Wesley, Reading (1999)
11. Logozzo, F.: Cibai: An Abstract Interpretation-Based Static Analyzer for Modular Analysis and Verfication of Java Classes. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, Springer, Heidelberg (2007)
12. Méndez, M., Navas, J., Hermenegildo, M.V.: An Efficient, Parametric Fixpoint Algorithm for Incremental Analysis of Java Bytecode. In: Proc. of the second workshop on Bytecode Semantics, Verification, Analysis and Transformation, Electronic Notes on Theoretical Computer Science. Braga, Portugal, March 2007. (to appear)
13. Pollet, I., Le Charlier, B., Cortesi, A.: Distinctness and Sharing Domains for Static Analysis of Java Programs. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 77–98. Springer, Heidelberg (2001)
14. Rossignoli, S., Spoto, F.: Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 95–110. Springer, Heidelberg (2005)
15. Sabelfeld, A., Myers, A.C.: Language-based Information-Flow Security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
16. Secci, S., Spoto, F.: Pair-Sharing Analysis of Object-Oriented Programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 320–335. Springer, Heidelberg (2005)
17. Spoto, F.: The JULIA Static Analyser (2007), `profs.sci.univr.it/~spoto/julia`
18. Spoto, F., Hill, P.M., Payet, E.: Path-Length Analysis for Object-Oriented Programs. In: Proc. of Emerging Applications of Abstract Interpretation, Vienna, Austria, March, `profs.sci.univr.it/~spoto/papers.html` (2006)
19. Spoto, F., Jensen, T.: Class Analyses as Abstract Interpretations of Trace Semantics. ACM Transactions on Programming Languages and Systems (TOPLAS) 25(5), 578–630 (2003)
20. Tarski, A.: A Lattice-theoretical Fixpoint Theorem and its Applications. Pacific J. Math. 5, 285–309 (1955)
21. Winskel, G.: The Formal Semantics of Programming Languages. MIT Press, Cambridge (1993)

# Author Index

Banterle, Francesco 315
Bauer, Jörg 249
Berdine, Josh 419
Brotherston, James 87

Calcagno, Cristiano 233, 402
Chang, Bor-Yuh Evan 384
Chen, Chung-Hsin 34
Chen, Kung 34
Chen, Yu 170
Clarke, Edmund 419
Cook, Byron 419
Cousot, Patrick 333

Delmas, David 437
Distefano, Dino 402

Ganty, Pierre 333
Giacobazzi, Roberto 315
Gopan, Denis 349
Goubault, Eric 137
Grumberg, Orna 69
Gupta, Aarti 366

Hardekopf, Ben 265

Ivančić, Franjo 366

Jeannet, Bertrand 52

Kamil, Amir 281
Khoo, Siau-Cheng 34
Kidd, Nicholas 200
King, Andy 121

Lal, Akash 200
Le Gall, Tristan 52
Leroux, Jérôme 184
Lin, Calvin 265

Magill, Stephen 419
Malkis, Alexander 218
Mao, Jian 170
Martel, Matthieu 298
Monniaux, David 104
Mycroft, Alan 18

Nandivada, V. Krishna 153
Necula, George C. 384

O'Hearn, Peter W. 402

Palsberg, Jens 153
Parkinson, Matthew 233
Payet, Étienne 452
Pereira, Fernando Magno Quintão 153
Podelski, Andreas 218
Putot, Sylvie 137

Raskin, Jean-François 333
Reps, Thomas 200, 349
Rival, Xavier 384
Rybalchenko, Andrey 218

Sankaranarayanan, Sriram 366
Shoham, Sharon 69
Simon, Axel 121
Souyris, Jean 437
Spoto, Fausto 452
Sutre, Grégoire 184

Tip, Frank 1
Touili, Tayssir 200

Vafeiadis, Viktor 233

Wang, Meng 34
Wei, Tao 170
Weng, Shu-Chun 34
Wilhelm, Reinhard 249

Yang, Hongseok 402
Yelick, Katherine 281

Zou, Wei 170