Chris Hankin
Igor Siveroni (Eds.)

LNCS 3672

Static Analysis

12th International Symposium, SAS 2005
London, UK, September 2005
Proceedings

Lecture Notes in Computer Science 3672

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

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
University of Dortmund, Germany
Madhu Sudan
Massachusetts Institute of Technology, MA, USA
Demetri Terzopoulos
New York University, NY, USA
Doug Tygar
University of California, Berkeley, CA, USA
Moshe Y. Vardi
Rice University, Houston, TX, USA
Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Chris Hankin Igor Siveroni (Eds.)

Static Analysis

12th International Symposium, SAS 2005
London, UK, September 7-9, 2005
Proceedings

Springer

Volume Editors

Chris Hankin
Igor Siveroni
Imperial College London, Department of Computing
180 Queen's Gate, London SW7 2BZ, UK
E-mail: {clh,siveroni}@doc.ic.ac.uk

Library of Congress Control Number: 2005931559

CR Subject Classification (1998): D.3.2-3, F.3.1-2, D.2.8, F.4.2, D.1

ISSN 0302-9743
ISBN-10 3-540-28584-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-28584-7 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springeronline.com

© Springer-Verlag Berlin Heidelberg 2005
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11547662 06/3142 5 4 3 2 1 0

Preface

Static analysis allows us to determine aspects of the dynamic behavior of programs and systems without actually executing them. Traditionally used in optimizing compilers, static analysis is now also used extensively in verification, software certification and semantics-based manipulation. The research community in static analysis covers a broad spectrum from foundational issues – new semantic models of programming languages and systems – through to practical tools. The series of Static Analysis Symposia has served as the primary venue for presentation and discussion of theoretical, practical and application advances in the area.

This volume contains the papers accepted for presentation at the 12th International Static Analysis Symposium (SAS 2005) which was held 7–9 September 2005 at Imperial College London. A total of 66 papers were submitted; the Program Committee held an online discussion which led to the selection of 22 papers for presentation. The selection was based on scientific quality, originality and relevance to the scope of SAS. Every paper was reviewed by at least 3 PC members or external referees. This volume also includes abstracts of talks given by the two invited speakers: Samson Abramsky FRS (University of Oxford) and Andrew Gordon (Microsoft Research, Cambridge).

On behalf of the Program Committee, the Program Chair would like to thank all of the authors who submitted papers and all of the external referees for their careful work in the reviewing process. The Program Chair would also particularly like to thank Igor Siveroni who provided local support for the conference management system and who helped in organizing the structure of this volume. We would also like to express our gratitude to Herbert Wiklicky and Bridget Gundry who masterminded the local arrangements.

SAS 2005 was held concurrently with *LOPSTR 2005*, the *International Symposium on Logic-Based Program Synthesis and Transformation*. We would like to thank Pat Hill (LOPSTR PC Chair) for her help and advice on the organizational aspects.

London, June 2005 Chris Hankin

Organization

Program Committee

Thomas Ball	Microsoft, USA
Radhia Cousot	CNRS/Ecole Polytechnique, France
Alessandra Di Pierro	Università di Pisa, Italy
Gilberto Filé	Università di Padova, Italy
Roberto Giacobazzi	Università di Verona, Italy
Chris Hankin (Chair)	Imperial College London, UK
Thomas Jensen	IRISA/CNRS Rennes, France
Andy King	University of Kent, UK
Pasquale Malacaria	Queen Mary College, UK
Laurent Mauborgne	École Normale Supérieure, France
Alan Mycroft	University of Cambridge, UK
Andreas Podelski	Max-Planck-Institut für Informatik, Germany
German Puebla	Technical University of Madrid, Spain
Ganesan Ramalingam	IBM, USA
Andrei Sabelfeld	Chalmers University of Technology, Sweden
Mooly Sagiv	Tel Aviv University, Israel
Harald Søndergaard	University of Melbourne, Australia
Bernhard Steffen	University of Dortmund, Germany

Steering Committee

Patrick Cousot	École Normale Supérieure, France
Gilberto Filé	Università di Padova, Italy
David Schmidt	Kansas State University, USA

Organizing Committee

Bridget Gundry
Igor Siveroni
Herbert Wiklicky

Referees

A. Askarov
G. Barthe
J. Bean
J. Berdine
S. Berezin
J. Bertrane
T. Harris
D. Hirsch
N. Kettle
R. Komondoor
A. Lawrence
O. Lee
X. Rival
F. Rossi
R. Rugina
O. Rüthing
P. Schmitt
R. Segala

F. Besson
B. Blanchet
F. Bueno
M. Carro
P. Caspi
O. Chitil
S. Chong
D. Clark
D. Colazzo
L. Colussi
J. Correas
S. Crafa
N. Dur
S. Edelkamp
C. Faggian
J. Feret
J. Field
A. Frisch
M. Gil
A. Gotlieb
T. Griffin
S. Gulwani
N. Halbwachs
R. Hansen
F. Levi
X. Li
F. Logozzo
A. Lokhmotov
R. Manevich
P. Manghi
J. Mariño
D. Massé
I. Mastreoni
H. Melgratti
A. Merlo
A. Miné
D. Monniaux
M. Müller-Olm
B. Nicolescu
K. Ostrovsky
L. Pareto
M. Preda
P. Pietrzak
H. Raffelt
F. Ranzato
A. Rensink
T. Rezk
N. Rinetzky
C. Segura
A. Simon
J Singer
J.-G. Smaus
F. Spoto
M. Strout
F. Tapparo
R. Thrippleton
S. Thompson
E. Tuosto
S. Valentini
A. Venet
L. Vigano
P. Wadler
H. Wiklicky
D. Xu
E. Yahav
G. Yorsh
S. Yong
E. Zaffanella
D. Zanardini
R. Zunino

Table of Contents

Invited Talks

Algorithmic Game Semantics and Static Analysis
Samson Abramsky 1

From Typed Process Calculi to Source-Based Security
Andrew D. Gordon 2

Contributed Papers

Widening Operators for Weakly-Relational Numeric Abstractions
Roberto Bagnara, Patricia M. Hill, Elena Mazzi, Enea Zaffanella 3

Generation of Basic Semi-algebraic Invariants Using Convex Polyhedra
Roberto Bagnara, Enric Rodríguez-Carbonell, Enea Zaffanella 19

Inference of Well-Typings for Logic Programs with Application to Termination Analysis
Maurice Bruynooghe, John Gallagher, Wouter Van Humbeeck 35

Memory Space Conscious Loop Iteration Duplication for Reliable Execution
Guilin Chen, Mahmut Kandemir, Mustafa Karakoy 52

Memory Usage Verification for OO Programs
Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, Martin Rinard . . . 70

Abstraction Refinement for Termination
Byron Cook, Andreas Podelski, Andrey Rybalchenko 87

Data-Abstraction Refinement: A Game Semantic Approach
Aleksandar Dimovski, Dan R. Ghica, Ranko Lazić 102

Locality-Based Abstractions
Javier Esparza, Pierre Ganty, Stefan Schwoon 118

Type-Safe Optimisation of Plugin Architectures
Neal Glew, Jens Palsberg, Christian Grothoff 135

Using Dependent Types to Certify the Safety of Assembly Code
Matthew Harren, George C. Necula 155

The PER Model of Abstract Non-interference
Sebastian Hunt, Isabella Mastroeni 171

A Relational Abstraction for Functions
Bertrand Jeannet, Denis Gopan, Thomas Reps 186

Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis
Yungbum Jung, Jaehwang Kim, Jaeho Shin, Kwangkeun Yi 203

Banshee: A Scalable Constraint-Based Analysis Toolkit
John Kodumal, Alex Aiken 218

A Generic Framework for Interprocedural Analysis of Numerical Properties
Markus Müller-Olm, Helmut Seidl 235

Finding Basic Block and Variable Correspondence
Iman Narasamdya, Andrei Voronkov 251

Boolean Heaps
Andreas Podelski, Thomas Wies 268

Interprocedural Shape Analysis for Cutpoint-Free Programs
Noam Rinetzky, Mooly Sagiv, Eran Yahav 284

Understanding the Origin of Alarms in ASTRÉE
Xavier Rival 303

Pair-Sharing Analysis of Object-Oriented Programs
Stefano Secci, Fausto Spoto 320

Exploiting Sparsity in Polyhedral Analysis
Axel Simon, Andy King 336

Secure Information Flow as a Safety Problem
Tachio Terauchi, Alex Aiken 352

Author Index 369

Algorithmic Game Semantics and Static Analysis

Samson Abramsky

Oxford University Computing Laboratory

Game Semantics has been developed over the past 12 years or so as a dist inc-tive approach to the semantics of programming language. It is composit ional in the tradition of denotational semantics, and has led to the cons truction of fully abstract models for programming languages incorporating a wide variety of features which have proved resistant to more tradition al approaches, including (combinations of): higher-order procedures, loca lly scoped variables and references, non-local control operators, non-det erminism, probability, concurrency and more. At the same time, game seman tics has a concrete aspect: programs are interpreted as strategies for ce rtain two-player games, and these strategies can be represented by automa ta. This algorithmic aspect of game semantics has been developed over the past few years, by Dan Ghica, Luke Ong, Andrzej Murawski and the present author. This has led to a novel approach to compositional model-checking and static analysis. We will survey some of the work which has been done , and discuss some directions for future research in this area.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, p. 1, 2005.
© Springer-Verlag Berlin Heidelberg 2005

From Typed Process Calculi to Source-Based Security

Andrew D. Gordon

Microsoft Research

The *source-based security* problem is to build tools to check security properties of the actual source code of a system, as opposed to some abstract model. Static analysis of C for buffer overruns is one approach. Another is to introduce *security types* as a programming language feature so that the typechecker proves security properties; for example, languages like Jif and Flow Caml can check noninterference properties of application-level code. Independently, security types have arisen in the setting of process calculi, for checking secrecy and authentication properties of abstract models of low-level cryptographic protocols, for instance.

My talk argues that recent developments in security types for process calculi can lead to better source-based security by typing. One development [2] removes a significant limitation of previous type systems and checks security in spite of the partial compromise of a dynamically-growing population of principals. Another [1] generalizes a type system for authentication to check authorization properties, by augmenting the typechecker with Datalog inference relative to a declarative authorization policy. Both developments rely on the idea of enriching process calculi with inert processes to represent both logical facts arising at runtime and also expected security invariants.

References

1. C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. In *European Symposium on Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 141–156. Springer, 2005.
2. A. D. Gordon and A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *CONCUR 2005—Concurrency Theory*, LNCS. Springer, 2005. To appear.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, p. 2, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Widening Operators for Weakly-Relational Numeric Abstractions*

Roberto Bagnara[1], Patricia M. Hill[2], Elena Mazzi[1], and Enea Zaffanella[1]

[1] Department of Mathematics, University of Parma, Italy
{bagnara, mazzi, zaffanella}@cs.unipr.it
[2] School of Computing, University of Leeds, UK
hill@comp.leeds.ac.uk

Abstract. We discuss the construction of proper widening operators on several weakly-relational numeric abstractions. Our proposal differs from previous ones in that we actually consider the semantic abstract domains, whose elements are *geometric shapes*, instead of the (more concrete) syntactic abstract domains of constraint networks and matrices. Since the closure by entailment operator preserves geometric shapes, but not their syntactic expressions, our widenings are immune from the divergence issues that could be faced by the previous approaches when interleaving the applications of widening and closure. The new widenings, which are variations of the *standard widening* for convex polyhedra defined by Cousot and Halbwachs, can be made as precise as the previous proposals working on the syntactic domains. The implementation of each new widening relies on the availability of an effective reduction procedure for the considered constraint description: we provide such an algorithm for the domain of *octagonal shapes*.

1 Introduction

Numerical properties are of great interest in the broad area of formal methods for their complete generality and since they often play a crucial role in the definition of static analyses and program verification techniques. In the field of abstract interpretation, classes of numerical properties are captured by numerical abstract domains. These have been and are widely used, either as the main abstraction for the application at hand, or as powerful ingredients to improve the precision of other abstract domains.

Among the wide spectrum of numerical abstractions proposed in the literature, the most famous ones are probably the (non-relational) abstract domain of intervals [16] and the (relational) abstract domain of convex polyhedra [19]. As far as the efficiency/precision trade-off is concerned, these domains occupy the opposite extremes of the spectrum: on the one hand, the operations on convex

* This work has been partly supported by MURST projects "Constraint Based Verification of Reactive Systems" and "AIDA — Abstract Interpretation: Design and Applications," and by a Royal Society (UK) International Joint Project (ESEP) award.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 3–18, 2005.
© Springer-Verlag Berlin Heidelberg 2005

polyhedra achieve a significant level of precision, which is however countered by a worst-case exponential time complexity, often leading to scalability problems; on the other hand, the great efficiency of the corresponding operations on intervals is made unappealing by the fact that the obtained precision is often unsatisfactory. This well-known dichotomy (which does not impede that, for some applications, convex polyhedra or intervals are the right choices) has motivated recent studies on several abstract domains that lie somehow between these two extremes, and can therefore be called *weakly-relational* abstract domains. Examples include domains based on constraint networks [3,4,5], the abstract domain of difference-bound matrices [25,32], the octagon abstract domain [26], the 'two variables per inequality' abstract domain [33], the octahedron abstract domain [15], and the abstract domain of template constraint matrices [31]. Moreover, similar proposals that are not abstractions of the domain of convex polyhedra have been put forward, including the abstract domain of bounded quotients [3] and the zone congruence abstract domain [27].

In this paper, we address the issue of the provision of proper widening operators for these domains. For the abstract domain of convex polyhedra, all the widenings that have been proposed are variations of, and/or improvements to, what is commonly referred to as the *standard widening* [19,22]. This is based on the general widening principle "drop the unstable components" applied to constraints. Not surprisingly, most proposals for widening operators for the weakly relational domains are based on the same principle and analogous to the standard widening. For instance, for the domain of difference bound matrices mentioned above, an operator meant to match the standard widening is given in [32]. Unfortunately, as pointed out in [25,26], this operator is not a widening, since it has no convergence guarantee. The reason is that *closure by entailment*, which is systematically performed so as to provide a canonical form for the elements and to improve the precision of several domain operations, has a negative interaction with the extrapolation operator of [32] that compromises the convergence guarantee. Intuitively, what can happen is that, while the extrapolation operator discards unstable constraints, the closure operation reinserts them (because they were redundant): failure to drop such unstable constraints can (and, in practice, quite often does) result in infinite upward iteration sequences. For this reason, it is proposed in [25,26] to apply the same operator given in [32] to the "syntactic" version of the same abstract domain, that is, where closure is only very carefully applied during the fixpoint computations.

We have taken a different approach and resolve the apparent conflict by considering a "semantic" abstract domain whose elements are the geometric shapes themselves. Since closure by entailment preserves the geometric shapes (even though this does not preserve their syntactic expressions), the approach is immune from the divergence problem described above. On the other hand, in order to use the standard widening as the basis of the proposed widening, it is important that we can compute *reduced* representations of the domain elements that encode non-redundant systems of constraints. Thus the implementations of any new widenings based on the semantic approach will need effective reduction

procedures for the considered constraint description: here we provide such an algorithm for the domain of *octagonal shapes*. As a by-product of our work on verifying the correctness of this reduction algorithm, we noticed that the algorithm for computing the strong closure of octagonal graphs as described in [25] could be simplified with a consequential improvement in its efficiency. This revised strong closure algorithm is also described here.

The paper is structured as follows: Section 2 recalls the required concepts and notations; Section 3 introduces the domain of bounded difference graphs; a domain of bounded difference shapes is presented in Section 4, where an alternative solution to the divergence problem is proposed; the generalization of the above results to the case of octagons is the subject of Section 5, where we define a new strong reduction procedure and an improved strong closure procedure for octagonal graphs, as well as a semantic widening operator for octagonal shapes. Section 6 concludes with a discussion of the results achieved. The proofs of all the stated results can be found in [6].

2 Preliminaries

The reader is assumed to be familiar with the fundamental concepts of lattice theory [13] and abstract interpretation theory [17,18]. We refer the reader to the classical works on the numeric domains of intervals [16] and convex polyhedra [19] for the specification of the corresponding widening operators.

Let $\mathbb{Q}_\infty := \mathbb{Q} \cup \{+\infty\}$ be totally ordered by the extension of '$<$' such that $d < +\infty$ for each $d \in \mathbb{Q}$. Let $\mathcal{N}$ be a finite set of *nodes*. A *weighted directed graph* (graph, for short) G in $\mathcal{N}$ is a pair $(\mathcal{N}, w)$, where $w \colon \mathcal{N} \times \mathcal{N} \to \mathbb{Q}_\infty$ is the weight function for G. A pair $(n_i, n_j) \in \mathcal{N} \times \mathcal{N}$ is an *arc* of G if $w(n_i, n_j) < +\infty$; the arc is *proper* if $n_i \neq n_j$.

A *path* $\pi = n_0 \cdots n_p$ in a graph $G = (\mathcal{N}, w)$ is a non-empty and finite sequence of nodes such that (n_{i-1}, n_i) is an arc of G, for all $i = 1, \ldots, p$; each arc (n_{i-1}, n_i) where $i = 1, \ldots, p$ is said to be *in* the path π. The path π is *proper* if all the arcs in it are proper. The path π is a *proper cycle* if it is a proper path and $n_0 = n_p$ (so that $p \geq 2$). The *length* of the path π is the number p of occurrences of arcs in π and denoted by $\|\pi\|$; the *weight* of the path π is $\sum_{i=1}^{p} w(n_{i-1}, n_i)$ and denoted by $w(\pi)$. The path π is a *zero-cycle* if it is a proper cycle with 0 weight. A graph is *consistent* if it has no negative weight cycles; it is *zero-cycle free* if all its proper cycles have strictly positive weights.

The set $\mathbb{G}$ of consistent graphs in $\mathcal{N}$ is partially ordered by the relation '$\trianglelefteq$' defined, for all $G_1 = (\mathcal{N}, w_1)$ and $G_2 = (\mathcal{N}, w_2)$, by

$$G_1 \trianglelefteq G_2 \quad \iff \quad \forall i, j \in \mathcal{N} : w_1(i,j) \leq w_2(i,j).$$

When augmented with a bottom element $\bot$ representing inconsistency, this partially ordered set becomes a (non-complete) lattice $\mathbb{G}_\bot = \langle \mathbb{G} \cup \{\bot\}, \trianglelefteq, \sqcap, \sqcup \rangle$, where '$\sqcap$' and '$\sqcup$' denote the (finitary) greatest lower bound and least upper bound operators, respectively.

Definition 1. (Closed graph.) *A consistent graph $G = (\mathcal{N}, w)$ is* closed *if the following properties hold:*

$$\forall i \in \mathcal{N} : w(i,i) = 0; \tag{1}$$

$$\forall i, j, k \in \mathcal{N} : w(i,j) \leq w(i,k) + w(k,j). \tag{2}$$

The (shortest-path) closure *of a consistent graph G in $\mathcal{N}$ is*

$$\operatorname{closure}(G) := \bigsqcup \bigl\{\, G^{\mathrm{c}} \in \mathbb{G} \bigm| G^{\mathrm{c}} \trianglelefteq G \textit{ and } G^{\mathrm{c}} \textit{ is closed} \,\bigr\}.$$

When trivially extended so as to behave as the identity function on the bottom element $\bot$, shortest-path closure is a kernel operator (monotonic, idempotent and reductive) on the lattice $\mathbb{G}_\bot$.

3 Systems of Bounded Differences

The typical way to simplify the domain of convex polyhedra is by restricting attention to particular subclasses of linear inequalities. One possibility, which has a long tradition in computer science [12], is to only consider *potential constraints*, also known as *bounded differences*: these are restricted to take the form $v_i - v_j \leq d$ or $\pm v_i \leq d$. Systems of bounded differences have been used by the artificial intelligence community as a way to reason about temporal quantities [2,20], as well as by the model checking community as an efficient yet precise way to model and propagate timing requirements during the verification of various kinds of concurrent systems [21,24]. In the abstract interpretation field, the idea of using an abstract domain of bounded differences was put forward in [3].

A finite system $\mathcal{C}$ of bounded differences on variables $\mathcal{V} = \{v_0, \ldots, v_{n-1}\}$ can be represented by a weighted directed graph $G = (\mathcal{N}_{\mathbf{0}}, w)$ where $\mathcal{N}_{\mathbf{0}} = \{\mathbf{0}\} \cup \mathcal{V}$, $\mathbf{0} \notin \mathcal{V}$ is the *special variable*, and the weight function w is defined, for each $v_i, v_j \in \mathcal{N}_{\mathbf{0}}$, by

$$w(v_i, v_j) := \begin{cases} \min\bigl\{\, d \in \mathbb{Q} \bigm| (v_i - v_j \leq d) \in \mathcal{C} \,\bigr\}, & \text{if } v_i \neq \mathbf{0} \text{ and } v_j \neq \mathbf{0}; \\ \min\bigl\{\, d \in \mathbb{Q} \bigm| (v_i \leq d) \in \mathcal{C} \,\bigr\}, & \text{if } v_i \neq \mathbf{0} \text{ and } v_j = \mathbf{0}; \\ \min\bigl\{\, d \in \mathbb{Q} \bigm| (-v_j \leq d) \in \mathcal{C} \,\bigr\}, & \text{if } v_i = \mathbf{0} \text{ and } v_j \neq \mathbf{0}; \\ 0, & \text{if } v_i = v_j = \mathbf{0}. \end{cases}$$

Notice that we assume that $\min \varnothing = +\infty$; moreover, unary constraints are encoded by means of the special variable, which is meant to always have value 0. A possible representation of (the weight function of) the graph G is by means of a matrix-like data structure called *Difference-Bound Matrix* (DBM) [12]. However, this representation provides no conceptual advantage over the isomorphic graph (or *constraint network* [20]) representation. For this reason we will consistently adopt the terminology and notation introduced in Section 2 for weighted directed graphs. In particular, a graph encoding a consistent system of bounded differences will be called a *Bounded Difference Graph* (BDG).

The first fully developed application of bounded differences in the field of abstract interpretation can be found in [32], where an abstract domain of closed BDGs is defined. In this case, the shortest-path closure requirement was meant as a simple and well understood way to obtain a canonical form for the domain elements by abstracting away from the syntactic details; since, basically, it corresponds to the *closure by entailment* of the encoded system of bounded differences. In [32] the specification of all the required abstract semantics operators is provided, including an operator that is meant to match the widening operators defined on more classical numeric domains. This operator can be interpreted either as a generalization for closed BDGs of the widening operator defined on the abstract domain of intervals [16], or as a restriction on the domain of closed BDGs of the standard widening defined on the abstract domain of convex polyhedra [19,22]: its implementation is based on the following upper bound operator on the set of consistent graph representations.

Definition 2. (Widening graphs.) *Let $G_1 = (\mathcal{N}, w_1)$ and $G_2 = (\mathcal{N}, w_2)$ be consistent graphs. Then $G_1 \nabla G_2 := (\mathcal{N}, w)$, where the weight function w is defined, for each $i, j \in \mathcal{N}$, by*

$$w(i,j) := \begin{cases} w_1(i,j), & \text{if } w_1(i,j) \geq w_2(i,j); \\ +\infty, & \text{otherwise.} \end{cases}$$

Unfortunately, as pointed out in [25,26], when used in conjunction with shortest-path closure, this extrapolation operator does not provide a convergence guarantee for fixpoint computations, hence it is not a widening. The reason is that, whereas the closure operation adds redundant constraints to the input BDG, a key requirement in the specification of the standard widening is that the first argument polyhedron must be described by a non-redundant system of constraints.[1] Thus we have a "conflict of interest" between the use of a convenient canonical form for the abstract domain —a form that also allows for increased precision of several domain operations— and the requirements of the widening.

The abstract domain of BDGs has been reconsidered in [25]. Differently from [32], in [25] BDGs are not required to be closed. In this more concrete, syntactic domain, the shortest-path closure operator maps each domain element into the smallest BDG encoding the same geometric shape. Closure is typically used as a preprocessing step before the application of most, though not all, of the abstract semantic operators, allowing for improved accuracy in the results of the abstract computation. The same widening operator proposed in [32] is also used in [25]; however, it is observed that this widening "could have intriguing interactions" with shortest-path closure, therefore identifying the divergence issue faced in [32]. This observation led the author of [25] to the adoption of the syntactic domain of BDGs, where closure is not enforced.

[1] This requirement was sometimes neglected in recent papers describing the standard widening on convex polyhedra; it was recently recalled and exemplified in [7,8]. Note that a similar requirement is implicitly present even in the specification of the widening on intervals.

4 Bounded Difference Shapes

While the analysis of the divergence problem is absolutely correct, the solution identified in [25] is sub-optimal since, as is usually the case, resorting to a syntactic domain (such as the one of BDGs) has a number of negative consequences, some of which will be recalled in Section 6.

To identify a simpler, more natural solution, we first have to acknowledge that an element of our abstract domain should be a geometric shape, rather than (any) one of its graph representations. To stress this concept, such an element will be called a *Bounded Difference Shape* (BDS). A BDS corresponds to the equivalence class of all the BDGs representing it. The implementation of the abstract domain can freely choose between these possible representations, switching at will from one to the other, as long as the semantic operators are implemented as expected. Notice that, in such a context, the shortest-path closure operator is just a transparent implementation detail: on the abstract domain of BDSs it corresponds to the identity function.

The other step towards the solution of the divergence problem is the simple observation that a BDS is a convex polyhedron and the set of all BDSs is closed under the application of the standard widening on convex polyhedra. Thus, no divergence problem can be incurred when applying the standard widening to an increasing sequence of BDSs. As mentioned in Section 3, a crucial requirement in the specification of the standard widening is that the first argument polyhedron is described by a non-redundant system of constraints [7,8]. Thus it is not surprising that using closed BDGs has problems since it is very likely that they will encode redundant constraints. By contrast, we propose the use of a maximal BDG in the equivalence class of BDGs representing the same geometric shape; since such a graph encodes no redundant constraints at all.

Definition 3. (Reduced graph.) *A consistent graph* G_1 *is* reduced *if, for each consistent graph* $G_2 \neq G_1$ *such that* $G_1 \trianglelefteq G_2$*, we have* $\mathrm{closure}(G_1) \neq \mathrm{closure}(G_2)$*. A* reduction *for the consistent graph* G *is any reduced graph* G_r *such that* $\mathrm{closure}(G) = \mathrm{closure}(G_\mathrm{r})$*.*

Hence, a graph is reduced if it is maximal in the subset of graphs having the same shortest-path closure. In order to provide a correct and reasonably efficient implementation of the standard widening on the domain of BDSs, all we need is a reduction procedure mapping a BDG representation into (any) one of the equivalent reduced graphs. Such an algorithm was defined in [24] and called *shortest-path reduction*. Basically, it is an extension of the transitive reduction algorithm of [1] to the case of weighted directed graphs. Note that, since each equivalence class may have many maximal elements, shortest-path reduction is not a properly defined operator on the domain of BDGs. However, the shortest-path reduction algorithm of [24] provides a canonical form as soon as we fix a total order for the nodes in the graph.

In summary, the solution to the divergence problem for BDSs is to apply the operator specified in Definition 2 to a reduced BDG representation of the first argument of the widening. From the point of view of the user, this will

be a transparent implementation detail: on the domain of BDSs, shortest-path reduction is the identity function, as was the case for shortest-path closure.

4.1 On the Precision of the Standard Widening

The standard widening on BDSs could result, if used with no precautions, in poorer precision with respect to its counterpart defined on the syntactic domain of BDGs. For increased precision, the specification of [25] prescribes two conditions that the abstract iteration sequence must satisfy:

1. the second argument of the widening should be represented by a closed BDG (note that, in this case, no divergence problem can arise);
2. the first BDG of the abstract iteration sequence $G_0 \trianglelefteq G_1 \trianglelefteq \ldots \trianglelefteq G_i \trianglelefteq \ldots$ should be closed too.

The effects of both improvements can be obtained also with the semantic domain of BDSs. As for the first one, this can be applied as is, leading to an implementation where the two arguments of the widening are represented by a reduced BDG and a closed BDG, respectively. The result of such a widening operator will depend on the specific reduced form computed for the first argument. The second precision improvement can be achieved by applying the well-known 'widening up to' technique defined in [23] or its variation called 'staged widening with thresholds' [14,29]: in practice, it is sufficient to add to the set of 'up to' thresholds all the constraints of the shortest-path closure of the first BDG G_0. Further precision improvements can be obtained by applying any delay strategy and/or the framework defined in [7,8].

5 Octagonal Graphs and Shapes

From a theoretical point of view, the observations made in the previous section are immediately applicable to any other weakly-relational numeric domain whose elements are convex polyhedra and is closed with respect to the application of the standard widening, therefore including the domains proposed in [15,26,31,33]. From a practical perspective, the success of such a construction depends on the availability of a reasonably efficient reduction procedure for the considered subclass of constraints, because the minimization algorithm for arbitrary linear inequality constraints is not efficient enough. In this section we provide such a reduction procedure for the *octagon* abstract domain [26].

The octagon abstract domain allows for the manipulation of *octagonal* constraints of the form $av_i + bv_j \leq c$, where $a, b \in \{-1, 0, +1\}$ (the same class of constraints was considered in [11], where octagons were called *simple sections*). Bounded differences can then be used to express octagonal constraints by splitting each variable $v_i \in \mathcal{V}$ into two forms: a positive form v_i^+, interpreted as $+v_i$; and a negative form v_i^-, interpreted as $-v_i$. Thus, an octagonal constraint such as $v_i + v_j \leq d$ can be translated into the bounded difference constraint $v_i^+ - v_j^- \leq d$; alternatively, the same constraint can be translated into $v_j^+ - v_i^- \leq d$. Note that

unary (octagonal) constraints such as $v_i \leq d$ and $-v_j \leq d$ can be encoded as $v_i^+ - v_i^- \leq 2d$ and $v_j^- - v_j^+ \leq 2d$, respectively, so that the special variable $\mathbf{0}$ is no longer needed.

In the following we assume that $\mathcal{N}^\pm = \{0, \ldots, 2n-1\}$ is a fixed and finite set of nodes where, for all $i = 0, \ldots, n-1$, the node $2i$ represents the positive form v_i^+ and $2i+1$ the negative form v_i^- of the variable v_i. Moreover, for all $i \in \mathcal{N}^\pm$, $\bar{\imath}$ denotes $i+1$ if i is even, and $i-1$ if i is odd. Thus, for all $i \in \mathcal{N}^\pm$, we also have $\bar{\imath} \in \mathcal{N}^\pm$ and $\bar{\bar{\imath}} = i$. Therefore, any finite system of octagonal constraints on the n variables $\mathcal{V} = \{v_0, \ldots, v_{n-1}\}$ can be represented by a weighted directed graph on the $2n$ nodes $\mathcal{N}^\pm$. Note that, for any $i, j \in \mathcal{N}^\pm$, as arcs (i, j) and $(\bar{\jmath}, \bar{\imath})$ denote equivalent expressions, the pair is said to be *coherent*. We restrict attention to consistent systems of constraints and hence to consistent graphs where coherent pairs of arcs have the same weight.

Definition 4. (Octagonal graph.) *An* octagonal graph *in $\mathcal{N}^\pm$ is any consistent graph $G = (\mathcal{N}^\pm, w)$ satisfying the coherence assumption:*

$$\forall i, j \in \mathcal{N}^\pm : w(i, j) = w(\bar{\jmath}, \bar{\imath}). \tag{3}$$

Thus any octagonal graph on the $2n$ nodes $\mathcal{N}^\pm$ encodes a consistent system of octagonal constraints on n variables. The set $\mathbb{O}$ of all octagonal graphs, with the usual addition of the bottom element representing the empty octagon, is a sub-lattice of $\mathbb{G}_\bot$, sharing the same least upper bound and greatest lower bound operators. Note that, at the implementation level, coherence can be automatically and efficiently enforced by letting arc (i, j) and arc $(\bar{\jmath}, \bar{\imath})$ share the same representation.

The octagon abstract domain developed in [26] is thus a syntactic domain having octagonal graphs as elements. When dealing with octagonal graphs, one has to remember the relation linking the positive and negative forms of each variable: in particular, besides transitivity, a proper closure by entailment procedure should also consider the following inference rule:

$$\frac{i - \bar{\imath} \leq d_1 \qquad \bar{\jmath} - j \leq d_2}{2(i - j) \leq d_1 + d_2} \tag{4}$$

Thus, the standard shortest-path closure algorithm is not enough to obtain a canonical form for octagonal graphs: to this end, a modified closure procedure is defined in [26], yielding *strongly closed* octagonal graphs.

Definition 5. (Strongly closed graph.) *An octagonal graph $G = (\mathcal{N}^\pm, w)$ is* strongly closed *if it is closed and the following property holds:*

$$\forall i, j \in \mathcal{N}^\pm : 2w(i, j) \leq w(i, \bar{\imath}) + w(\bar{\jmath}, j). \tag{5}$$

The strong closure *of an octagonal graph G in $\mathcal{N}^\pm$ is*

$$\mathrm{Closure}(G) := \bigsqcup \bigl\{\, G^{\mathrm{C}} \in \mathbb{O} \bigm| G^{\mathrm{C}} \trianglelefteq G \text{ and } G^{\mathrm{C}} \text{ is strongly closed} \,\bigr\}.$$

Similarly to shortest-path closure, strong closure is a kernel operator on the lattice of octagonal graphs.

By repeating the reasoning of the previous section, we define the semantic abstract domain of *octagonal shapes*, whose elements are equivalence classes of octagonal graphs representing the same geometric shape. Hence, strong closure maps an octagonal graph representation of a non-empty octagonal shape into the minimum element of the corresponding equivalence class. The dual procedure, mapping the octagonal graph into (any) one of the maximal elements in its equivalence class, is called *strong reduction*.

Definition 6. (Strongly reduced graph.) *An octagonal graph* G_1 *is* strongly reduced *if, for each octagonal graph* $G_2 \neq G_1$ *such that* $G_1 \trianglelefteq G_2$*, we have* $\mathrm{Closure}(G_1) \neq \mathrm{Closure}(G_2)$*. A* strong reduction *for the octagonal graph* G *is any strongly reduced octagonal graph* G_R *such that* $\mathrm{Closure}(G) = \mathrm{Closure}(G_\mathrm{R})$*.*

Note that, in the above definition, we only compare G_1 with other *octagonal* graphs. Thus, we explicitly disregard those trivial redundancies that are due to the coherence assumption. This is not a real problem because, as discussed before, any reasonable implementation will automatically and efficiently filter away these kinds of redundancies.

5.1 A Strong Reduction Procedure for Octagonal Graphs

In this section we generalize the shortest-path reduction algorithm of [24] so as to obtain a strong reduction procedure for octagonal graphs. Clearly, the algorithm of [24] cannot be used without modifications, since it takes no account of the redundancies caused by the new constraint inference rule (4). Nonetheless, the high-level structure of the strong reduction procedure is the same as that defined in [24] for shortest-path reduction:

1. Compute the closure by entailment of the constraint graph;
2. Partition the nodes into equivalence classes based on equality constraints;
3. Decompose the graph so as to separate those arcs that link different equivalence classes (encoding only inequalities) from the partition information (encoding the equivalence classes themselves, i.e., all the equalities);
4. Reduce the subgraph that gives constraints on different equivalence classes;
5. Reduce the partition information;
6. Merge the results of steps 4 and 5 to obtain the reduced constraint graph.

We now describe each of the above steps, formally stating the correctness of the overall procedure.

Step 1 of the algorithm can be performed by applying the strong closure procedure defined in [26].

Step 2 is also easily implemented by observing that, in a strongly closed octagonal graph, equality constraints correspond to proper zero-cycles having length two.

Definition 7. (Zero-equivalence.) *Let $G = (\mathcal{N}^{\pm}, w)$ be a strongly closed octagonal graph. The nodes $i, j \in \mathcal{N}^{\pm}$ are* zero-equivalent *in G, denoted $i \equiv_G j$, if and only if $w(i,j) = -w(j,i)$.*

While step 6 carries over from BDGs to octagonal graphs, the formal definition of steps 3–5 of the reduction algorithm is more difficult for octagonal graphs than it was for BDGs, as it requires some understanding of the structure of the zero-equivalence classes.

As a first observation, note that $i \equiv_G j$ if and only if $\bar{\imath} \equiv_G \bar{\jmath}$. Therefore, if $\mathcal{E} \subseteq \mathcal{N}^{\pm}$ is a zero-equivalence class for the strongly closed octagonal graph G, then $\overline{\mathcal{E}} := \{\, \bar{\imath} \in \mathcal{N}^{\pm} \mid i \in \mathcal{E} \,\}$ is also a zero-equivalence class for G. We say that $\mathcal{E}$ is *non-singular* if $\mathcal{E} \cap \overline{\mathcal{E}} = \varnothing$, and *singular* if $\mathcal{E} = \overline{\mathcal{E}}$; there is at most one singular zero-equivalence class in G. We associate to each zero-equivalence class $\mathcal{E} \subseteq \mathcal{N}^{\pm}$ a *leader* $\ell_{\mathcal{E}} := \min \mathcal{E}$; the class having the leader in positive (resp., negative) form will be said to be a positive (resp., negative) zero-equivalence class. Thus, the singular zero-equivalence class, if present, is always positive and, for non-singular zero-equivalence classes $\mathcal{E}$ and $\overline{\mathcal{E}}$, we have $\ell_{\overline{\mathcal{E}}} = \overline{\ell_{\mathcal{E}}}$.

We are now ready to provide a formal specification for step 3 of the strong reduction algorithm. As was the case in [24], the first subgraph resulting from the decomposition, relating nodes in different zero-equivalence classes, is obtained by only connecting the leaders. However, we do not connect the leader of the singular zero-equivalence class to the other leaders. The second subgraph only encodes those constraints relating nodes in the same zero-equivalence class.

Definition 8. (Non-singular leaders and zero-equivalence subgraphs.) *Let $G = (\mathcal{N}^{\pm}, w)$ be a strongly closed octagonal graph and $\mathcal{L} \subseteq \mathcal{N}^{\pm}$ the set of leaders of the non-singular zero-equivalence classes for G. The* non-singular leaders' subgraph *of G is the graph $L = (\mathcal{N}^{\pm}, w_{\mathrm{L}})$, where the weight function w_{L} is defined, for each $i, j \in \mathcal{N}^{\pm}$, by*

$$w_{\mathrm{L}}(i,j) := \begin{cases} w(i,j), & \text{if } i = j \text{ or } \{i,j\} \subseteq \mathcal{L}; \\ +\infty, & \text{otherwise.} \end{cases}$$

The zero-equivalence subgraph *of G is the graph $E = (\mathcal{N}^{\pm}, w_{\mathrm{E}})$, where the weight function w_{E} is defined, for each $i, j \in \mathcal{N}^{\pm}$, by*

$$w_{\mathrm{E}}(i,j) := \begin{cases} w(i,j), & \text{if } i \equiv_G j; \\ +\infty, & \text{otherwise.} \end{cases}$$

Step 4 of the strong reduction algorithm is implemented by checking, for each proper arc in the non-singular leaders' subgraph, whether it can be obtained from the other arcs by a single application of the constraint inference rules. Once again, note that we disregard redundancies caused by the coherence assumption.

Definition 9. (Strongly atomic arc and subgraph.) *Let $G = (\mathcal{N}^{\pm}, w)$ be an octagonal graph. An arc (i,j) of G is* atomic *if it is proper and, for all*

$k \in \mathcal{N}^{\pm} \setminus \{i, j\}$, $w(i,j) < w(i,k) + w(k,j)$. *The arc* (i,j) *is* strongly atomic *if it is atomic and either* $i = \bar{\jmath}$ *or* $2w(i,j) < w(i,\bar{\imath}) + w(\bar{\jmath}, j)$.

The strongly atomic subgraph *of* G *is the graph* $A = (\mathcal{N}^{\pm}, w_{\mathrm{A}})$ *where the weight function* w_{A} *is defined, for all* $i, j \in \mathcal{N}^{\pm}$, *by*

$$w_{\mathrm{A}}(i,j) = \begin{cases} w(i,j), & \text{if } (i,j) \text{ is strongly atomic in } G; \\ +\infty, & \text{otherwise.} \end{cases}$$

The implementation of step 5 of the algorithm, i.e., the strong reduction of the zero-equivalence subgraph, is performed by reducing each zero-equivalence class in isolation. Once again, we exploit the total ordering defined on $\mathcal{N}^{\pm}$.

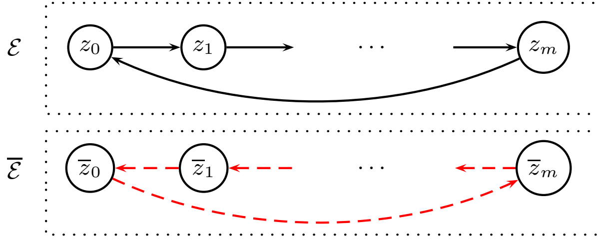


Fig. 1. Strong reduction for non-singular zero-equivalence classes

The strong reduction for a positive non-singular zero-equivalence class $\mathcal{E}$ follows that of [24]: it creates a single zero-cycle connecting all nodes in $\mathcal{E}$ following their total ordering, where the weights of the component arcs are as in the strong closure of the graph. By the coherence assumption, the nodes in the corresponding negative zero-equivalence class $\overline{\mathcal{E}}$ are automatically connected in the opposite order. Figure 1 shows the arcs in the strong reduction of both $\mathcal{E}$ and $\overline{\mathcal{E}}$, where $\mathcal{E} = \{z_0, \ldots, z_m\}$ is the positive class and where $z_0 < \cdots < z_m$. The strong reduction for a singular zero-equivalence class $\mathcal{E}$ is similar except that there is now a single zero-cycle connecting all the positive and negative nodes in $\mathcal{E}$. Figure 2 shows the strong reduction for the singular zero-equivalence class $\mathcal{E} = \{z_0, \overline{z}_0, \ldots, z_m, \overline{z}_m\}$, where $z_0 < \overline{z}_0 < \cdots < z_m < \overline{z}_m$. In both Figures 1 and 2, the dashed arcs are those that can be obtained from the non-dashed ones by application of the coherence assumption.

The following definition formalizes the above observations.

Definition 10. (Zero-equivalence reduction.) *Let* $G = (\mathcal{N}^{\pm}, w)$ *be a strongly closed octagonal graph and let* w' *be the weight function defined, for all* $i, j \in \mathcal{N}^{\pm}$, *as follows: if* $i, j \in \mathcal{E}$ *for some positive zero-equivalence class* $\mathcal{E}$ *of* G *and*

– *if* $\mathcal{E} = \{z_0, \ldots, z_m\}$ *is non-singular, assuming* $z_0 < \cdots < z_m$,

$$w'(i,j) := \begin{cases} w(i,j), & \text{if } i = z_{h-1},\ j = z_h, \text{ for some } h = 1\ \ldots, m; \\ w(i,j), & \text{if } i = z_m,\ j = z_0 \text{ and } m > 0; \\ +\infty, & \text{otherwise;} \end{cases}$$

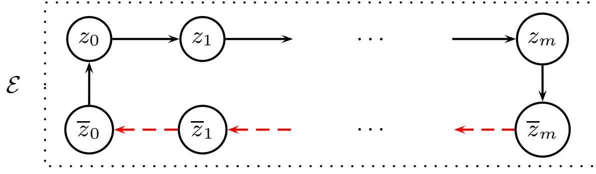


Fig. 2. Strong reduction for the singular zero-equivalence class

– *if* $\mathcal{E} = \{z_0, \overline{z}_0, \dots, z_m, \overline{z}_m\}$ *is singular, assuming* $z_0 < \overline{z}_0 < \cdots < z_m < \overline{z}_m$,

$$w'(i,j) := \begin{cases} w(i,j), & \textit{if } i = z_{h-1},\ j = z_h, \textit{ for some } h = 1 \dots, m; \\ w(i,j), & \textit{if } i = \overline{z}_0,\ j = z_0 \textit{ or } i = z_m,\ j = \overline{z}_m; \\ +\infty, & \textit{otherwise;} \end{cases}$$

and $w'(i,j) := +\infty$*, otherwise. Then, the* zero-equivalence reduction *for* G *is the octagonal graph* $Z = (\mathcal{N}^{\pm}, w_{\mathrm{Z}})$*, where, for each* $i, j \in \mathcal{N}^{\pm}$*,*

$$w_{\mathrm{Z}}(i,j) := \min\bigl\{w'(i,j), w'(\overline{\jmath}, \overline{\imath})\bigr\}.$$

The final step 6 of the strong reduction algorithm is implemented by computing the greatest lower bound $A \sqcap Z$, where A is the strongly atomic subgraph of L and Z is the zero-equivalent reduction of E, as obtained at steps 4 and 5 of the algorithm.

Theorem 1. *Given an octagonal graph* G*, the strong reduction algorithm computes a strong reduction for* G*.*

If n is the cardinality of the original set $\mathcal{V}$ of variables, then steps 1 and 4 of the algorithm have worst-case complexity in $O(n^3)$, while all the others steps are in $O(n^2)$. Thus, the overall procedure has cubic complexity. As was the case for the reduction procedure of [24], once the ordering of variables is fixed, the strong reduction algorithm returns a canonical form for octagonal graphs.

5.2 An Improved Strong Closure Algorithm

The formal proof of Theorem 1 led to a new result regarding the strong closure operator for octagonal graphs. The strong closure algorithm formalized in [26,30] performs n local propagation steps: in each step, a rather involved variant of the constraint propagation in the Floyd-Warshall algorithm is followed by another constraint propagation corresponding to the new inference rule (4). A finely tuned implementation of this algorithm [28] performs $20n^3 + 24n^2$ coefficient operations (additions and comparisons), where n is the dimension of the vector space. It turns out that the interleaving of the two kinds of propagation steps is not needed: the same final result can be obtained by the application of the classical Floyd-Warshall closure algorithm followed by a *single* local propagation step using the constraint inference rule (4).

Theorem 2. *Let $G^{c} = (\mathcal{N}^{\pm}, w^{c})$ be a closed octagonal graph. Consider the graph $G^{S} = (\mathcal{N}^{\pm}, w^{S})$, where w^{S} is defined, for each $i, j \in \mathcal{N}^{\pm}$, by*

$$w^{S}(i,j) := \min\bigl\{w^{c}(i,j), w^{c}(i,\bar{\imath})/2 + w^{c}(\bar{\jmath},j)/2\bigr\}.$$

Then $G^{S} = \mathrm{Closure}(G^{c})$.

By coupling the above optimization with the classical Floyd-Warshall algorithm, we obtain a much simpler implementation performing $16n^3 + 4n^2 + 4n$ coefficient operations: the saving is always above 20% and it is above 30% for $n \leq 8$.

5.3 A Semantic Widening for Octagonal Shapes

A correct implementation of the standard widening on octagonal shapes is obtained by computing any strong reduction of the octagonal graph representing the first argument. As in the case of BDSs, for maximum precision the strongly closed representation for the second argument should be computed. Even better, by adopting the following minor variant, we obtain a "truly semantic" widening operator for the domain of octagonal shapes.

Definition 11. (Widening octagonal shapes.) *Let $S_1, S_2 \in \wp(\mathbb{R}^n)$, where $\varnothing \neq S_1 \subseteq S_2$, be two octagonal shapes represented by the strongly reduced octagonal graph G_1 and the strongly closed octagonal graph G_2, respectively. Let also $S \in \wp(\mathbb{R}^n)$ be the octagonal shape represented by the octagonal graph $G_1 \nabla G_2$. Let* $\dim(T)$ *denote the affine dimension of shape T. Then we define*

$$S_1 \nabla S_2 := \begin{cases} S_2, & \text{if } \dim(S_1) < \dim(S_2); \\ S, & \text{otherwise.} \end{cases}$$

By refraining from applying the graph-based widening when the affine dimension of the geometric shapes is increasing, the operator becomes independent from the specific strongly reduced form computed, i.e., from the total ordering defined on the nodes of the graphs. Also note that the test $\dim(S_1) < \dim(S_2)$ can be efficiently decided by checking whether the nodes of the two octagonal graphs are partitioned into different collections of zero-equivalence classes.

Theorem 3. *The operator '∇' of* Definition 11 *is a proper widening on the domain of octagonal shapes. Let '∇_{s}' be the standard widening on the domain of convex polyhedra, as defined in* [22]*. Then, for all octagonal shapes $S_1, S_2 \in \mathbb{R}^n$ such that $\varnothing \neq S_1 \subseteq S_2$, we have $S_1 \nabla S_2 \subseteq S_1 \nabla_{\mathrm{s}} S_2$.*

The definition of a semantic widening for the domain of BDSs is obtained by simply replacing, in Definition 11, the strongly reduced and strongly closed octagonal graph representations with the reduced and closed BDG representations, respectively. Then a result similar to Theorem 3 holds for BDSs.

6 Conclusion

By considering the semantic abstract domains of geometric shapes, instead of their syntactic representations in terms of constraint networks, we have shown how proper widening operators can be derived for several weakly-relational numeric abstractions. For what concerns the efficient representation of octagonal shapes by means of octagonal graphs, we have specified and proved correct a strong reduction procedure, as well as a more efficient strong closure procedure.

It is worth stressing that both the syntactic and the semantic abstract domains are well defined and may be safely adopted for the implementation of a static analysis application. Nonetheless, it can be argued that using a semantic abstract domain provides several advantages, as already pointed out in [25, Section 5] where the domain of BDGs is compared to the domain of closed BDGs.[2] For instance, it is noted that the domain of closed BDGs allows for the specification of a nicer, injective meaning function; also, the least upper bound operator on BDGs is not the most precise approximation of the union of two geometric shapes. In summary, the discussion in [25, Section 5] makes clear that the solution to the divergence problem for the abstract iteration sequence was the one and only motivation for adopting a syntactic domain.

One disadvantage of syntactic abstract domains concerns the user-level interfaces of the corresponding software implementations. Namely, the user of a syntactic abstract domain (e.g., the developer of a specific static analysis application using this domain) has to be aware of many details that, in principle, should be hidden by the implementation. As an example, consider the shortest-path closure and reduction procedures for BDGs, which the user might rightfully see as semantics-preserving operations. As a matter of fact, for the syntactic domain of BDGs, these are not semantics-preserving: their application affects both the precision and the convergence of the abstract iteration. In such a situation, the documentation of the abstract domain software needs to include several warnings about the correct usage of these operators, so as to avoid possible pitfalls. In contrast, when adopting the semantic domain of BDSs, both the closure and reduction operators may be excluded from the public interface while the implementation can apply them where and when needed or appropriate. Such an approach is systematically pursued in the implementation of the *Parma Polyhedra Library* [10] (PPL, `http://www.cs.unipr.it/ppl`), free software distributed under the GNU General Public License; future releases of the library will support computations on the domains of BDSs and octagonal shapes.

Another potential drawback of the adoption of a syntactic abstract domain can be found in the application of domain refinement operators. As an example, consider the application of the *finite powerset operator* [9] to the domains of BDGs and BDSs, so as to obtain two abstract domains that are able to represent finite disjunctions of the corresponding abstract elements. In both cases, by providing the widenings on BDGs and BDSs with appropriate finite convergence certificates [9], it will be possible to lift them to corresponding widenings on the

[2] Similar observations, tailored to the case of octagons, are also in [26, Section VII].

powerset domains. However, when upgrading the syntactic domain, avoidable redundancies will be incurred, since different disjuncts inside a domain element may represent the same geometric shape; furthermore, these "duplicates" cannot be systematically removed, since by doing so we could change the value of the finite convergence certificate of the powerset element, possibly breaking the convergence guarantee of the lifted widening.

The shortest-path reduction algorithm of [24] has also been considered in the PhD thesis of A. Miné [30] as a tool for the computation of *hollow* (i.e., sparse) representations for BDGs, as originally proposed in [24], so as to obtain memory space savings. The author appears not to identify the positive interaction between reduction and widening and, as a consequence, he conjectures that the computation of hollow representations could compromise the convergence of the abstract iteration sequence (see [30, Section 3.8.2]). An adaptation of the reduction algorithm for the case of octagonal graphs is defined in [30, Section 4.5.2]: this differs from the one proposed in Section 5.1 and may fail to obtain a strongly reduced graph in the sense of Definition 6.

The theoretical results concerning weighted directed graphs hold when the data type adopted for the representation of weights allows for exact computations. If a floating-point data type is considered, then most of these results will be broken due to rounding errors, so that the implementation of a truly semantic abstract domain will not be possible. Nonetheless, the (approximate) reduction operators allow for the removal of most of the syntactic redundancies.

References

1. A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.
2. J. F. Allen and H. A. Kautz. A model of naive temporal reasoning. In *Formal Theories of the Commonsense World*, pp. 251–268. Ablex, Norwood, NJ, 1985.
3. R. Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages*. PhD thesis, Dipartimento di Informatica, Università di Pisa, Italy, 1997.
4. R. Bagnara, R. Giacobazzi, and G. Levi. Static analysis of CLP programs over numeric domains. In *Proc. WSA 1992*, vol. 81–82 of *Bigre*, pp. 43–50, Bordeaux.
5. R. Bagnara, R. Giacobazzi, and G. Levi. An application of constraint propagation to data-flow analysis. In *Proc. CAIA 1993*, pp. 270–276, Orlando, FL.
6. R. Bagnara, P. M. Hill, E. Mazzi, and E. Zaffanella. Widening operators for weakly-relational numeric abstractions. Quaderno 399, Dipartimento di Matematica, Univ. di Parma, Italy, 2005. Available at `http://www.cs.unipr.it/Publications/`.
7. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In *Proc. SAS 2003*, vol. 2694 of *LNCS*, pp. 337–354, San Diego.
8. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 2005. To appear.
9. R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. In *Proc. VMCAI 2004*, vol. 2937 of *LNCS*, pp. 135–148, Venice, Italy.
10. R. Bagnara, P. M. Hill, and E. Zaffanella. *The Parma Polyhedra Library User's Manual*. Department of Mathematics, University of Parma, release 0.7, 2004.

11. V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proc. PLDI 1989*, vol. 24(7) of *ACM SIGPLAN Notices*, pp. 41–53, Portland, OR.
12. R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.
13. G. Birkhoff. *Lattice Theory.* American Mathematical Society, 3rd edition, 1967.
14. B. Blanchet, P. Cousot, R. Cousot, J. Feret *et al.*, A static analyzer for large safety-critical software. In *Proc. PLDI 2003*, pp. 196–207, San Diego, CA.
15. R. Clarisó and J. Cortadella. The octahedron abstract domain. In *Proc. SAS 2004*, vol. 3148 of *LNCS*, pp. 312–327, Verona, Italy.
16. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. ISOP 1976*, pp. 106–130, Paris, France.
17. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL 1977*, pp. 238–252, New York.
18. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. POPL 1979*, pp. 269–282, New York.
19. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL 1978*, pp. 84–96, Tucson, AR.
20. E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32(3):281–331, 1987.
21. D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proc. AVMFSS 1989*, vol. 407 of *LNCS*, pp. 197–212, Grenoble, France.
22. N. Halbwachs. *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme.* PhD thesis, Université de Grenoble, France, 1979.
23. N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Form. Method Syst. Des.*, 11(2):157–185, 1997.
24. K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *Proc. RTSS 1997*, pp. 14–24, San Francisco, CA.
25. A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Proc. PADO 2001*, vol. 2053 of *LNCS*, pp. 155–172, Aarhus, Denmark.
26. A. Miné. The octagon abstract domain. In *Proc. WCRE'01*, pp. 310–319, Stuttgart.
27. A. Miné. A few graph-based relational numerical abstract domains. In *Proc. SAS 2002*, vol. 2477 of *LNCS*, pp. 117–132, Madrid, Spain.
28. A. Miné. *The Octagon Abstract Domain Library.* École Normale Supérieure, Paris, France, release 0.9.6, 2002. Available at `http://www.di.ens.fr/~mine/oct/`.
29. A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *Proc. ESOP 2004*, vol. 2986 of *LNCS*, pp. 3–17, Barcelona, Spain.
30. A. Miné. *Weakly Relational Numerical Abstract Domains.* PhD thesis, École Polytechnique, Paris, France, 2005.
31. S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Proc. VMCAI 2005*, pp. 25–41, Paris, France.
32. R. Shaham, E. K. Kolodner, and S. Sagiv. Automatic removal of array memory leaks in Java. In *Proc. CC 2000*, vol. 1781 of *LNCS*, pp. 50–66, Berlin, Germany.
33. A. Simon, A. King, and J. M. Howe. Two variables per linear inequality as an abstract domain. In *Proc. LOPSTR 2002*, vol. 2664 of *LNCS*, pp. 71–89, Madrid.

Generation of Basic Semi-algebraic Invariants Using Convex Polyhedra*

Roberto Bagnara[1], Enric Rodríguez-Carbonell[2], and Enea Zaffanella[1]

[1] Department of Mathematics, University of Parma, Italy
{bagnara, zaffanella}@cs.unipr.it
[2] Software Department, Technical University of Catalonia, Spain
erodri@lsi.upc.edu

Abstract. A technique for generating invariant polynomial *inequalities* of bounded degree is presented using the abstract interpretation framework. It is based on overapproximating basic semi-algebraic sets, i.e., sets defined by conjunctions of polynomial inequalities, by means of convex polyhedra. While improving on the existing methods for generating invariant polynomial *equalities*, since polynomial inequalities are allowed in the guards of the transition system, the approach does not suffer from the prohibitive complexity of the methods based on quantifier-elimination. The application of our implementation to benchmark programs shows that the method produces non-trivial invariants in reasonable time. In some cases the generated invariants are essential to verify safety properties that cannot be proved with classical linear invariants.

1 Introduction

The discovery of invariant properties is at the core of the analysis and verification of infinite state systems such as sequential programs and reactive systems. For this reason, invariant generation has been a major research problem since the seventies. Abstract interpretation [11] provides a solid foundation for the development of techniques automatizing the synthesis of invariants of several classes, most significantly intervals [10], linear equalities [20] and linear inequalities [14].

For some applications, linear invariants are not enough to get a precise analysis of numerical programs and nonlinear invariants may be needed as well. For example, the ASTRÉE static analyzer, which has been successfully employed to verify the absence of run-time errors in flight control software [13], implements the ellipsoid abstract domain [7], which represents a certain class of quadratic inequality invariants. Moreover, it has been acknowledged elsewhere [28,30] that nonlinear invariants are sometimes required to prove program properties.

As a consequence, a remarkable amount of work has been recently directed to the generation of invariant *polynomial equalities*. Some of the methods plainly

* This work has been partially supported by PRIN project "AIDA — Abstract Interpretation: Design and Applications," by the "LogicTools" project (CICYT TIN 2004-03382), and the FPU grant AP2002-3693 from the Spanish MEC.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 19–34, 2005.
© Springer-Verlag Berlin Heidelberg 2005

ignore all the conditional guards [25,27]; other methods can only consider the polynomial equalities in the guards [8,31], whereas some other proposals [23,26] can handle *polynomial disequalities* in guards (i.e., guards of the form $p \neq 0$ where p is a polynomial). None of the techniques previously mentioned can handle the case of *polynomial inequalities* in the guards: these are ignored to the expense of precision.

In this paper we present a method for generating conjunctions of polynomial inequalities as invariants of transition systems, which we have chosen as our programming model. The transition systems that the approach can handle admit finite conjunctions of polynomial inequalities as guards and initial conditions, as well as polynomial assignments and nondeterministic assignments where the *rvalue* is unknown (these may correspond, for instance, to the assignment of expressions that cannot be modeled by means of polynomials).

Formally, our technique is an abstract interpretation in the lattice of *polynomial cones* of bounded degree, which are the algebraic structures analogous to vector spaces in the context of polynomial equality invariants [8]. Intuitively, the approach is based on considering nonlinear terms as additional independent variables and using convex polyhedra to represent polynomial cones in this extended set of variables. In order to reduce the loss of precision induced by this overapproximation, additional linear constraints are added conservatively to the polyhedra, so as to enforce some (semantically redundant) nonlinear constraints that would be lost in the translation. The strength of the approach is that, while allowing for a much broader class of programs than linear analysis, it uses the very same underlying machinery: this permits the adoption of already existing implementations of convex polyhedra like [4], as well as the possibility of resorting to further approximations, such as *bounded differences* [1] or *octagons* [22], when facing serious scalability problems.

The rest of the paper is organized as follows. In the next subsection, related work is briefly reviewed. Section 2 gives background information on algebraic geometry, transition systems and abstract interpretation. Section 3 presents the main contribution of the paper, where it is shown how polynomial inequalities can be discovered as invariants by means of polynomial cones, represented as convex polyhedra. The experimental evaluation of our implementation of these ideas is described in Section 4. Finally in Section 5 we summarize the contributions of the paper and sketch some ideas for future work.

1.1 Related Work

To the best of our knowledge, the first contribution towards the generation of invariant polynomial inequalities is [6]. The authors consider a simple class of transition systems, where assignments are of the form $x := x + k$ or $x := k$ with $k \in \mathbb{Z}$. Such a transition system is soundly abstracted into a new one whose exact reachability set is computable and overapproximates the reachability set of the original system. Besides the fact that the programming model is more restrictive than the one used in this paper, these ideas do not seem to have

undergone experimental evaluation so that, as far as we can tell, their practical value remains to be assessed.

In [19], Kapur proposes a method based on imposing that a template polynomial inequality with undetermined coefficients is invariant and solving the resulting constraints over the coefficients by real quantifier elimination. Unfortunately, the great computational complexity of quantifier elimination appears to make the method impractical: as the author reports, an experimental implementation performed poorly or did not return any answer for all the analyzed programs [D. Kapur, personal communication, 2005].

A similar idea is at the core of [9,28], where, instead of real quantifier elimination, semidefinite programming is employed. The method, which is reported to perform rather efficiently for several interesting cases, automatically determines *one* solution to the constraint system on the template parameters. This is particularly appropriate for proving program termination because, once a class of candidate ranking functions has been chosen, any solution belonging to this class is good enough. The same approach has also been applied to the computation of invariant properties. In this case, according to [9], the one above becomes the main limitation of the method: any invariant property, even a weak one, may be obtained and it is unclear whether it is possible to drive the solver so as to produce a more precise invariant in the same class.

In [30], Sankaranarayanan et al. propose a technique for generating linear invariants by linear programming. It is based on imposing, as invariants, constraints where the coefficients of the variables are fixed *a priori*; the analysis then returns, for each such constraint, an independent term for which the constraint is indeed an invariant of the system (in the case where this is not possible, the analysis returns $\pm\infty$). A generalization of this approach for the discovery of invariant polynomial inequalities by means of semidefinite programming is sketched. Similarly, the ellipsoid abstract domain [7] allows to generate invariant quadratic inequalities with two variables by also fixing the coefficients of terms and leaving the independent term to be determined by the analysis. The approach proposed in this paper differs in that we do not need to fix any of these coefficients in advance, but rather it is the analysis itself that determines all coefficients.

2 Preliminaries

2.1 Algebraic Geometry

We denote the real numbers by $\mathbb{R}$, and the nonnegative real numbers by $\mathbb{R}_+$. A *term* in the tuple of variables $\boldsymbol{x} = (x_1, \ldots, x_n)$ is any expression of the form $\boldsymbol{x}^{\boldsymbol{\alpha}} := x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n}$, where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \in \mathbb{N}^n$. A *monomial* is an expression of the form $c \cdot \boldsymbol{x}^{\boldsymbol{\alpha}}$, simply written as $c\boldsymbol{x}^{\boldsymbol{\alpha}}$, where $c \in \mathbb{R}$ and $\boldsymbol{x}^{\boldsymbol{\alpha}}$ is a term. The *degree* of a monomial $c\boldsymbol{x}^{\boldsymbol{\alpha}}$ with $c \neq 0$ is $\deg(c\boldsymbol{x}^{\boldsymbol{\alpha}}) := \alpha_1 + \cdots + \alpha_n$; the degree of 0 is $\deg(0) := -\infty$. A *polynomial* is a finite sum of monomials. The set of all polynomials in $\boldsymbol{x}$ with coefficients in $\mathbb{R}$ is denoted by $\mathbb{R}[\boldsymbol{x}]$. In the following we will only consider polynomials in *canonical form*, meaning that all the

monomials occurring in them have non-null coefficients and distinct terms. The degree of a non-null polynomial is the maximum of the degrees of its monomials. We denote by $\mathbb{R}_d[\boldsymbol{x}]$ the set of all polynomials in $\mathbb{R}[\boldsymbol{x}]$ having degree at most d. In particular, the polynomials in $\mathbb{R}_1[\boldsymbol{x}]$, i.e., having degree at most 1, are called *linear*; similarly, the polynomials in $\mathbb{R}_2[\boldsymbol{x}]$ are called *quadratic*.

A *polynomial equality* (resp., *polynomial inequality*) is a formula of the form $p = 0$ (resp., $p \geq 0$), where $p \in \mathbb{R}[\boldsymbol{x}]$. Both will be referred to as *polynomial constraints* or simply *constraints*. Given a constraint system ψ, i.e., a finite set of polynomial constraints, we define

$$\operatorname{poly}(\psi) := \bigl\{\, p \in \mathbb{R}[\boldsymbol{x}] \bigm| (p = 0) \in \psi \text{ or } (-p = 0) \in \psi \text{ or } (p \geq 0) \in \psi \,\bigr\}.$$

We will sometimes abuse notation by writing the set ψ to denote the finite conjunction of the constraints occurring in it.

The *algebraic set* defined by a finite set of polynomials $\{p_1, \ldots, p_k\} \subseteq \mathbb{R}[\boldsymbol{x}]$ is the set of points that satisfy the corresponding polynomial equalities, i.e., $\bigl\{\, \boldsymbol{v} \in \mathbb{R}^n \bigm| p_1(\boldsymbol{v}) = 0, \ldots, p_k(\boldsymbol{v}) = 0 \,\bigr\}$. Similarly, the *basic semi-algebraic set* defined by the same set of polynomials is the set of points that satisfy all the corresponding polynomial inequalities: $\bigl\{\, \boldsymbol{v} \in \mathbb{R}^n \bigm| p_1(\boldsymbol{v}) \geq 0, \ldots, p_k(\boldsymbol{v}) \geq 0 \,\bigr\}$.

2.2 Transition Systems

In this section we define our programming model: *transition systems*.

Definition 1. (Transition system.) *A* transition system $(\boldsymbol{x}, \mathcal{L}, \mathcal{T}, \mathcal{I})$ *is a tuple that consists of the following components:*

- *An n-tuple of real-valued* variables $\boldsymbol{x} = (x_1, \ldots, x_n)$.
- *A finite set $\mathcal{L}$ of* locations.
- *A finite set $\mathcal{T} \subset \mathcal{L} \times \mathcal{L} \times \wp(\mathbb{R}^n) \times \bigl(\mathbb{R}^n \rightarrow \wp(\mathbb{R}^n)\bigr)$ of* transitions. *A transition $(\ell, \ell', \gamma, \rho) \in \mathcal{T}$ consists of a* source location $\ell \in \mathcal{L}$, *a* target location $\ell' \in \mathcal{L}$, *a* guard $\gamma \in \wp(\mathbb{R}^n)$ *that enables the transition, and, finally, an* update map $\rho \colon \mathbb{R}^n \rightarrow \wp(\mathbb{R}^n)$ *that relates the values of the variables before and after the firing of the transition.*
- *A map $\mathcal{I} \colon \mathcal{L} \rightarrow \wp(\mathbb{R}^n)$ from locations to* initial conditions.

The guards, the update maps and the initial conditions are all assumed to be finitely computable.

The state of a transition system is completely characterized by the location at which control resides and by a valuation for the variables.

Definition 2. (Local and global state.) *A* local state *(at some unspecified location) is any real vector $\boldsymbol{v} = (v_1, \ldots, v_n) \in \mathbb{R}^n$, interpreted as a valuation for the variables $\boldsymbol{x} = (x_1, \ldots, x_n)$: in local state $\boldsymbol{v}$, we have $x_i = v_i$ for each $i = 1$, $\ldots$, n. A* global state *is a pair $(\ell, \boldsymbol{v})$, where $\ell \in \mathcal{L}$ and $\boldsymbol{v}$ is the local state at ℓ.*

Definition 3. (Run, initial state.) *A* run *of the transition system* $(\boldsymbol{x}, \mathcal{L}, \mathcal{T}, \mathcal{I})$ *is a sequence of global states* $(\ell_0, \boldsymbol{v}_0)$, $(\ell_1, \boldsymbol{v}_1)$, $(\ell_2, \boldsymbol{v}_2)$, $\ldots$ *such that* (1) $(\ell_0, \boldsymbol{v}_0)$ *is an* initial state, *that is* $\boldsymbol{v}_0 \in \mathcal{I}(\ell_0)$, *and* (2) *for each pair of consecutive states,* $(\ell_i, \boldsymbol{v}_i)$ *and* $(\ell_{i+1}, \boldsymbol{v}_{i+1})$, *there exists a transition* $(\ell_i, \ell_{i+1}, \gamma, \rho) \in \mathcal{T}$ *that is enabled, i.e.,* $\boldsymbol{v}_i \in \gamma$, *and such that* $\boldsymbol{v}_{i+1} \in \rho(\boldsymbol{v}_i)$.

The fundamental notion is that of an *invariant* of a transition system:

Definition 4. (Reachable state, invariant property and map.) *A global state* $(\ell, \boldsymbol{v})$ *is called* reachable *in the transition system* $S = (\boldsymbol{x}, \mathcal{L}, \mathcal{T}, \mathcal{I})$, *if there exists a run* $(\ell_0, \boldsymbol{v}_0)$, $(\ell_1, \boldsymbol{v}_1)$, $\ldots$, $(\ell_m, \boldsymbol{v}_m)$ *of* S *such that* $(\ell, \boldsymbol{v}) = (\ell_m, \boldsymbol{v}_m)$. *We denote the set of reachable states of* S *by* $\mathrm{reach}(S)$, *and the set of (local) reachable states at location* ℓ, *i.e., those* $\boldsymbol{v}$ *such that* $(\ell, \boldsymbol{v}) \in \mathrm{reach}(S)$, *by* $\mathrm{reach}_\ell(S)$.

If $\boldsymbol{x} = (x_1, \ldots, x_n)$, *an* invariant property *of* S *at location* $\ell \in \mathcal{L}$ *(also called an* invariant*) is any set* $I \in \wp(\mathbb{R}^n)$ *such that* $\mathrm{reach}_\ell(S) \subseteq I$. *Finally, an* invariant map *is a map* $\mathrm{inv} \colon \mathcal{L} \to \wp(\mathbb{R}^n)$ *such that for any* $\ell \in \mathcal{L}$, $\mathrm{inv}(\ell)$ *is an invariant of* S *at location* ℓ.

In this paper we focus on a particular class of transition systems, *basic semi-algebraic transition systems*:

Definition 5. (Basic semi-algebraic transition system.) *A transition system* $(\boldsymbol{x}, \mathcal{L}, \mathcal{T}, \mathcal{I})$, *where* $\boldsymbol{x} = (x_1, \ldots, x_n)$, *is called* basic semi-algebraic *if:*

1. *for all* $(\ell, \ell', \gamma, \rho) \in \mathcal{T}$, γ *is a basic semi-algebraic set and there exist* $k \leq n$ *polynomials* $p_1, \ldots, p_k \in \mathbb{R}[\boldsymbol{x}]$ *and distinct indices* $i_1, \ldots, i_k \in \{1, \ldots, n\}$ *such that* $\rho(\boldsymbol{v}) = \bigl\{ (v'_1, \ldots, v'_n) \in \mathbb{R}^n \bigm| v'_{i_1} = p_1(\boldsymbol{v}), \ldots, v'_{i_k} = p_k(\boldsymbol{v}) \bigr\}$ *for each* $\boldsymbol{v} \in \mathbb{R}^n$;
2. $\mathcal{I}(\ell)$ *is a basic semi-algebraic set, for each* $\ell \in \mathcal{L}$.

Notice that a basic semi-algebraic transition system can also model *nondeterministic assignments*, that is, assignments whose *rvalue* is unknown.

Example 1. The program shown on the left of Figure 1 is a minor variant of the program in [15, p. 64], computing the floor of the square root of a natural number a. The basic semi-algebraic transition system shown on the right of the figure models the (second loop of the) program. Note that even the original program in [15], which has the disequality $c \neq 1$ in the guard of the second loop, can be modeled as a basic semi-algebraic transition system (by translating $c \neq 1$ as $c \leq 0 \vee c \geq 2$ and then having four transitions instead of two). The variant in Figure 1 has been adopted just for presentation purposes: its analysis leads to the same invariants that are computed when analyzing the original program.

2.3 Abstract Interpretation

Abstract interpretation [11] is a general theory of approximation of the behavior of dynamic discrete systems. One of its classical applications is the inference of invariant properties of transition systems [12]. This is done by specifying the set

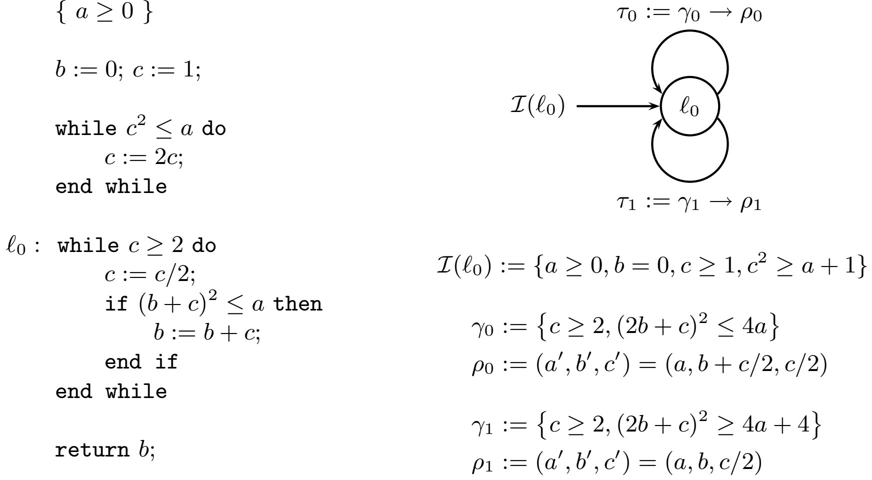


Fig. 1. A program and its model as a basic semi-algebraic transition system

of reachable states of the given transition system as the solution of a system of fixpoint equations. The concrete behavior of the transition system is then over-approximated by setting up a corresponding system of equations defined over an *abstract domain*, providing computable representations for the *abstract properties* that are of interest for the analysis, as well as *abstract operations* that are sound approximations of the *concrete operations* used by the transition system being analyzed. An approximation of one solution of the system of abstract equations can be found iteratively, possibly applying further conservative approximations and using convergence acceleration methods, such as *widenings* [11]. One of the main advantages of this methodology for the inference of invariant properties is that the correctness of the obtained results follows by design.

More specifically, given a transition system $S = (\boldsymbol{x}, \mathcal{L}, \mathcal{T}, \mathcal{I})$, the set of its reachable states $\mathrm{reach}(S)$ can be characterized by means of the following system of fixpoint equations where, for each $\ell \in \mathcal{L}$, we have the equation

$$\mathrm{reach}_\ell(S) = \mathcal{I}(\ell) \cup \bigcup \Bigl\{ \rho\bigl(\mathrm{reach}_{\ell'}(S) \cap \gamma\bigr) \Bigm| (\ell', \ell, \gamma, \rho) \in \mathcal{T} \Bigr\}. \tag{1}$$

The least fixpoint of this system of equations, with respect to the pointwise extension of the subset ordering on $\wp(\mathbb{R}^n)$, is $\mathrm{reach}(S)$; any overapproximation of $\mathrm{reach}(S)$ yields an invariant map for S.

3 Approximating Basic Semi-algebraic Sets

The construction of our abstract domain is analogous to that in [8], where *pseudo-ideals* of polynomials are introduced to infer polynomial *equalities* as invariants, while still reasoning in the framework of linear algebra. Here, we extend this approach so as to handle polynomial *inequalities* as invariants.

In [8], the basic underlying definition is that of a *vector space* of polynomials:

Definition 6. (Vector space.) *A set of polynomials $V \subseteq \mathbb{R}[\boldsymbol{x}]$ is a* vector space *if* (1) $0 \in V$*; and* (2) $\lambda p + \mu q \in V$ *whenever* $p, q \in V$ *and* $\lambda, \mu \in \mathbb{R}$*. For each $Q \subseteq \mathbb{R}[\boldsymbol{x}]$, the* vector space spanned by Q*, denoted by $\mathcal{V}(Q)$, is the least vector space containing Q, that is,*

$$\mathcal{V}(Q) := \left\{ \sum_{i=1}^{s} \lambda_i q_i \in \mathbb{R}[\boldsymbol{x}] \;\middle|\; s \in \mathbb{N}, \forall i \in \{1, \ldots, s\} : \lambda_i \in \mathbb{R}, q_i \in Q \right\}.$$

Given a vector space V, we associate the constraint $p = 0$ to any $p \in V$. Notice that, if $p, q \in \mathbb{R}[\boldsymbol{x}]$ and $\boldsymbol{v} \in \mathbb{R}^n$ are such that $p(\boldsymbol{v}) = 0$ and $q(\boldsymbol{v}) = 0$, then $(\lambda p + \mu q)(\boldsymbol{v}) = 0$, for any $\lambda, \mu \in \mathbb{R}$. Further, for any $\boldsymbol{v} \in \mathbb{R}^n$, the zero polynomial trivially satisfies $0(\boldsymbol{v}) = 0$. Thus, the set of polynomials that evaluate to 0 on a set of states $S \in \wp(\mathbb{R}^n)$, that is $\{ p \in \mathbb{R}[\boldsymbol{x}] \mid \forall \boldsymbol{v} \in S : p(\boldsymbol{v}) = 0 \}$, has the structure of a vector space. Unfortunately, this vector space has infinite dimension. In order to work with objects of finite dimension, it is necessary to approximate by bounding the degrees of the polynomials.

Moreover, when considering polynomials as elements of a vector space, the algebraic relationships between terms such as x_1, x_2 and x_1x_2 are lost. For instance, consider the vector space $\mathcal{V}\bigl(\{x_1, x_2 - x_1x_2\}\bigr)$, generated by the polynomial equalities $x_1 = 0$ and $x_2 = x_1x_2$. Then, even though the polynomial equality $x_2 = 0$ is semantically entailed by the previous ones, $x_2 \notin \mathcal{V}\bigl(\{x_1, x_2 - x_1x_2\}\bigr)$. The reason is that the vector space generated by x_1 and $x_2 - x_1x_2$ only includes the *linear* combinations of its generators, whereas in the case above x_2 can only be obtained by a *nonlinear* combination of the generators, namely $x_2 = x_2 \cdot (x_1) + 1 \cdot (x_2 - x_1x_2)$. This problem can be solved by adding the polynomial x_1x_2 to the set of generators, so that the polynomial $x_2 \in \mathcal{V}\bigl(\{x_1, x_1x_2, x_2 - x_1x_2\}\bigr)$ can be obtained by the linear combination $0 \cdot (x_1) + 1 \cdot (x_1x_2) + 1 \cdot (x_2 - x_1x_2)$.

In general, in order to reduce the loss of precision due to the linearization of the abstraction, additional polynomials are added taking into account that, when $p \in \mathbb{R}[\boldsymbol{x}]$ and $\boldsymbol{v} \in \mathbb{R}^n$ are such that $p(\boldsymbol{v}) = 0$, we have $(pq)(\boldsymbol{v}) = 0$ for each $q \in \mathbb{R}[\boldsymbol{x}]$. Therefore, pseudo-ideals are defined as follows:

Definition 7. (Pseudo-ideal.) *A* pseudo-ideal of degree $d \in \mathbb{N}$ *is a vector space $P \subseteq \mathbb{R}_d[\boldsymbol{x}]$ such that $pq \in P$ whenever $p \in P$, $q \in \mathbb{R}[\boldsymbol{x}]$ and* $\deg(pq) \leq d$*. For each $Q \subseteq \mathbb{R}_d[\boldsymbol{x}]$, the* pseudo-ideal of degree d spanned by Q*, denoted by* $\mathrm{pseudo}_d(Q)$*, is the least pseudo-ideal of degree d containing Q.*

Pseudo-ideals are closed under addition, product by scalars and degree-bounded product by polynomials. For instance, for the example above $x_1x_2 \in \mathrm{pseudo}_2\bigl(\{x_1, x_2\}\bigr)$. Pseudo-ideals are the elements of the abstract domain used in [8].

In order to extend this methodology to the generation of invariant polynomial inequalities, a first, necessary step is the identification, in the basic semi-algebraic context, of an adequate algebraic structure playing the same role of vector spaces for polynomial equalities. It turns out that *polynomial cones* are the right notion:

Definition 8. (Polynomial cone.) *A set of polynomials* $C \subseteq \mathbb{R}[\boldsymbol{x}]$ *is a* polynomial cone *if* (1) $1 \in C$*; and* (2) $\lambda p + \mu q \in C$ *whenever* $p, q \in C$ *and* $\lambda, \mu \in \mathbb{R}_+$. *For each* $Q \subseteq \mathbb{R}[\boldsymbol{x}]$*, the* polynomial cone generated by Q*, denoted by* $\mathcal{C}(Q)$*, is the least polynomial cone containing* Q*, that is,*

$$\mathcal{C}(Q) := \left\{ \lambda + \sum_{i=1}^{s} \lambda_i q_i \in \mathbb{R}[\boldsymbol{x}] \;\middle|\; \lambda \in \mathbb{R}_+, s \in \mathbb{N}, \forall i \in \{1, \ldots, s\} : \lambda_i \in \mathbb{R}_+, q_i \in Q \right\}.$$

Mimicking the reasoning done before for vector spaces, we associate the constraint $p \geq 0$ to any polynomial p in the polynomial cone C. Consider the basic semi-algebraic set defined by the constraint system

$$\psi = \{f_1 = 0, \ldots, f_h = 0, g_1 \geq 0, \ldots, g_k \geq 0\} \tag{2}$$

where, for each $i = 1, \ldots, h$ and $j = 1, \ldots, k$, we have $f_i, g_j \in \mathbb{R}[\boldsymbol{x}]$. Then, the set of polynomial inequalities that are consequences of ψ define a polynomial cone. Indeed, $\psi \implies (1 \geq 0)$ trivially; and, if $\psi \implies (p \geq 0)$ and $\psi \implies (q \geq 0)$, clearly $\psi \implies (\lambda p + \mu q \geq 0)$ for each $\lambda, \mu \in \mathbb{R}_+$. As was the case for the vector space of polynomials, this set of polynomials has infinite dimension. In order to deal with objects of finite dimension, we again fix an upper bound for the degrees of the polynomials. Moreover, to mitigate the precision loss due to linearization, we close this cone with respect to degree-bounded product by polynomials. Thus, the analog of pseudo-ideals in the basic semi-algebraic setting are *product-closed polynomial cones*:

Definition 9. (Product-closed polynomial cone.) *A* product-closed polynomial cone of degree $d \in \mathbb{N}$ *is a polynomial cone* $C \subseteq \mathbb{R}_d[\boldsymbol{x}]$ *satisfying:*

(1) $pq \in C$ *whenever* $\{p, -p\} \subseteq C$, $q \in \mathbb{R}[\boldsymbol{x}]$ *and* $\deg(pq) \leq d$*;*
(2) $pq \in C$ *whenever* $p, q \in C$ *and* $\deg(pq) \leq d$.

For each $Q \subseteq \mathbb{R}_d[\boldsymbol{x}]$*, the* product-closed polynomial cone of degree d generated by Q*, denoted by* $\mathrm{prod}_d(Q)$*, is the least product-closed polynomial cone of degree* d *containing* Q*.*

Let ψ be a constraint system defining a basic semi-algebraic set. Then, once the degree bound $d \in \mathbb{N}$ is fixed, ψ can be abstracted by the product-closed polynomial cone $\mathrm{prod}_d\bigl(\mathrm{poly}(\psi) \cap \mathbb{R}_d[\boldsymbol{x}]\bigr)$. The approximation forgets those polynomials occurring in ψ having a degree greater than d. Also note that the precision of the approximation depends on the specific constraint system ψ.

The abstraction is clearly sound. For the linear case it is also complete. In fact, consider any finite set of linear constraints $\varphi = \{p_1 \geq 0, \ldots, p_k \geq 0\}$, which we assume to be satisfiable. Then, the corresponding product-closed polynomial cone of degree 1 is $L = \mathrm{prod}_1\bigl(\mathrm{poly}(\varphi)\bigr) = \mathcal{C}\bigl(\mathrm{poly}(\varphi)\bigr)$, whose elements are the nonnegative linear consequences of φ. On the other hand, if $p \in \mathbb{R}_1[\boldsymbol{x}]$ is a linear polynomial such that $\varphi \implies (p \geq 0)$, then by Farkas' lemma there exists $\boldsymbol{\mu} = (\mu_0, \ldots, \mu_k) \in \mathbb{R}_+^{k+1}$ such that $p = \mu_0 + \sum_{i=1}^{k} \mu_i p_i$; in other words, $p \in L$.

In the general nonlinear setting, the abstraction constituted by product-closed polynomial cones is not complete. Notice however that the set of *all* invariant polynomial inequalities is not computable in basic semi-algebraic transition systems. Worse, the set of all invariant *linear equalities* is not computable in transition systems even if restricted to linear equality guards [24].

3.1 Representation

Given a finitely generated polynomial cone, by exploiting classical duality results [33], each polynomial generator p is interpreted as the constraint $p \geq 0$; these polynomial constraints are then linearized so as to define a convex polyhedron on an extended ambient space. The linearization in the abstraction process implies that all terms are considered as different variables. For instance, in Example 1, the terms a, b, c, c^2 are all regarded as different and potentially independent variables, and the initial condition $\mathcal{I}(\ell_0) = \{a \geq 0, b = 0, c \geq 1, c^2 \geq a+1\}$ is interpreted as defining, by means of 4 constraints, a convex polyhedron in an ambient space of dimension at least 4. In general, given a transition system on an n-tuple $\boldsymbol{x}$ of variables and a degree bound d, the introduction of the auxiliary variables, standing for all the nonlinear terms of degree at most d, yields an m-tuple $\boldsymbol{y}$ of variables, where each y_i corresponds to one of the $m = \binom{n+d}{d} - 1$ different terms $\boldsymbol{x}^{\boldsymbol{\alpha}} \in \mathbb{R}_d[\boldsymbol{x}]$, where $\boldsymbol{\alpha} \neq \boldsymbol{0}$. Thus, computation in the abstract domain of cones of degree d is only feasible provided d is small, e.g., 2 or 3. In the following, we will denote each y_i by writing the corresponding term $\boldsymbol{x}^{\boldsymbol{\alpha}}$.

It remains to be seen how the linearized constraint system can be closed, according to Definition 9, with respect to bounded-degree product by polynomials. Rather than trying to obtain the exact product-closed polynomial cone by means of a potentially expensive fixpoint computation, we actually approximate it as follows. Consider the constraint system ψ as defined in (2). Let $\mathcal{M}(g_1, \ldots, g_k)$ be the *multiplicative monoid* generated by the g_j's, i.e., the set of finite products of g_j's including 1 (the empty product). Let us consider the polynomials $p = \sum_{i=1}^{h} r_i f_i + \sum_j \lambda_j g'_j$, where for each $i = 1, \ldots, h$, $r_i \in \mathbb{R}[\boldsymbol{x}]$ is such that $\deg(r_i f_i) \leq d$, and for each j, $\lambda_j \in \mathbb{R}_+$ and $g'_j \in \mathcal{M}(g_1, \ldots, g_k) \cap \mathbb{R}_d[\boldsymbol{x}]$. These polynomials belong to $\mathrm{prod}_d\bigl(\mathrm{poly}(\psi) \cap \mathbb{R}_d[\boldsymbol{x}]\bigr)$, and thus soundly overapproximate the basic semi-algebraic set corresponding to ψ. Algorithm enrich_d, given in Figure 2, computes this approximation, which in practice provides comparable precision to the product closure at much less computational cost.

Example 2. Consider Example 1. The application of procedure enrich_2 to the initial condition $\phi = \mathcal{I}(\ell_0)$ yields the system of constraints

$$\begin{aligned}
\phi' &= \mathrm{enrich}_2\bigl(\phi \cap \mathbb{R}_2[\boldsymbol{x}]\bigr)\\
&= \mathrm{enrich}_2\bigl(\{b = 0\} \cup \{a \geq 0, c \geq 1, c^2 \geq a+1\}\bigr)\\
&= \{b = 0, ab = 0, b^2 = 0, bc = 0\}\\
&\qquad \cup \{a \geq 0, c \geq 1, c^2 \geq a+1, a^2 \geq 0, c^2 \geq 1, ac \geq 0\}.
\end{aligned}$$

Require: A finite set of polynomial equalities $\varphi = \{f_1 = 0, \dots, f_h = 0\}$ and a finite set of polynomial inequalities $\psi = \{g_1 \geq 0, \dots, g_k \geq 0\}$.
Ensure: $\varphi' = \{f'_1 = 0, \dots, f'_{h'} = 0\}$ and $\psi' = \{g'_1 \geq 0, \dots, g'_{k'} \geq 0\}$ are finite sets of polynomial equalities and inequalities, respectively, such that $\mathrm{poly}(\varphi' \cup \psi') \subseteq \mathbb{R}_d[\boldsymbol{x}]$ and $\mathcal{C}\bigl(\mathrm{poly}(\varphi \cup \psi) \cap \mathbb{R}_d[\boldsymbol{x}]\bigr) \subseteq \mathcal{C}\bigl(\mathrm{poly}(\varphi' \cup \psi')\bigr) \subseteq \mathrm{prod}_d\bigl(\mathrm{poly}(\varphi \cup \psi) \cap \mathbb{R}_d[\boldsymbol{x}]\bigr)$.

$\varphi' := \psi' := \emptyset$
for all $(f = 0) \in \varphi$ **do**
  if $\deg(f) \leq d$ **then**
    for all $\boldsymbol{x}^{\boldsymbol{\alpha}}$ **such that** $\deg(\boldsymbol{x}^{\boldsymbol{\alpha}}) \leq d - \deg(f)$ **do**
      $\varphi' := \varphi' \cup \{\boldsymbol{x}^{\boldsymbol{\alpha}} f = 0\}$
for all finite product g' of g's **such that** $(g \geq 0) \in \psi$ **do**
  if $\deg(g') \leq d$ **then**
    $\psi' := \psi' \cup \{g' \geq 0\}$

Fig. 2. Algorithm enrich_d

In this case, $\mathcal{C}\bigl(\mathrm{poly}(\phi')\bigr) = \mathrm{prod}_2\bigl(\mathrm{poly}(\phi) \cap \mathbb{R}_2[\boldsymbol{x}]\bigr)$. In general, a precision loss may occur; for instance, letting $\chi = \{x \geq 0, x^2 \geq 0, y - y^2 \geq 0, y^2 \geq 0\}$, we have $\chi = \mathrm{enrich}_2(\chi)$, but $(xy \geq 0) \in \mathrm{prod}_2\bigl(\mathrm{poly}(\chi)\bigr) \setminus \mathcal{C}\bigl(\mathrm{poly}(\chi)\bigr)$.

3.2 Abstract Semantics

In this section we review the operations required in order to perform abstract interpretation of transition systems using polynomial cones as abstract values.

Union. Given two (finitely generated) polynomial cones C_1 and C_2 representing the polynomial constraint systems ψ_1 and ψ_2, respectively, we would like to approximate the union of the corresponding basic semi-algebraic sets using another basic semi-algebraic set. By duality, this amounts to computing the intersection cone $C_1 \cap C_2$: for each $p \in C_1 \cap C_2$ and $\boldsymbol{v} \in \mathbb{R}^n$ such that $\psi_1(\boldsymbol{v}) \vee \psi_2(\boldsymbol{v})$, either $\psi_1(\boldsymbol{v})$, so that $p(\boldsymbol{v}) \geq 0$ as $p \in C_1$; or $\psi_2(\boldsymbol{v})$, so that $p(\boldsymbol{v}) \geq 0$ as $p \in C_2$. Thus, the approximation is sound. At the implementation level, since polynomial cones are represented by means of their (linearized) duals, this intersection of cones corresponds to the convex polyhedral hull operation.

Intersection. Given two (finitely generated) polynomial cones $C_1 = \mathcal{C}(Q_1)$ and $C_2 = \mathcal{C}(Q_2)$, we would like to compute the intersection of the respective basic semi-algebraic sets. Then a sound approximation is to compute the cone spanned by the union of the generators, $\mathcal{C}(Q_1 \cup Q_2)$. In order to reduce the loss of precision due to linearization, we enrich this cone with respect to degree-bounded product by polynomials as explained above. Thus, the polynomial cone corresponding to the intersection is $\mathrm{enrich}_d(Q_1 \cup Q_2)$.

Update. Each basic semi-algebraic update map $\rho\colon \mathbb{R}^n \to \wp(\mathbb{R}^n)$, defined over the original n-tuple of variables $\boldsymbol{x}$, is approximated by a *linearized* update map $\rho^\star\colon \mathbb{R}^m \to \wp(\mathbb{R}^m)$, where $m = \binom{n+d}{d} - 1$, defined over the extended m-tuple $\boldsymbol{y}$ of terms. The new update map $\rho^\star$ is obtained by composing a sequence of simpler maps, each one approximating the effect of ρ on a single term. For the sake of notation, if variable y_i corresponds to term $\boldsymbol{x}^{\boldsymbol{\alpha}}$ and $p \in \mathbb{R}_d[\boldsymbol{x}]$, let $\boldsymbol{x}^{\boldsymbol{\alpha}} \mapsto p$ denote the deterministic update map such that, for each $\boldsymbol{w} \in \mathbb{R}^m$,

$$(\boldsymbol{x}^{\boldsymbol{\alpha}} \mapsto p)(\boldsymbol{w}) := \Big\{ \big(w_1, \ldots, w_{i-1}, p(\boldsymbol{w}), w_{i+1}, \ldots, w_m\big) \Big\} \subseteq \mathbb{R}^m. \tag{3}$$

Note that the (possibly nonlinear) polynomial $p \in \mathbb{R}_d[\boldsymbol{x}]$ on the original tuple of variables is interpreted as a linear polynomial $p \in \mathbb{R}_1[\boldsymbol{y}]$ on the extended ambient space, so that Equation (3) indeed defines an affine map. Similarly, $\boldsymbol{x}^{\boldsymbol{\alpha}} \mapsto ?$ denotes the nondeterministic update map such that, for each $\boldsymbol{w} \in \mathbb{R}^m$,

$$(\boldsymbol{x}^{\boldsymbol{\alpha}} \mapsto ?)(\boldsymbol{w}) := \big\{ (w_1, \ldots, w_{i-1}, u, w_{i+1}, \ldots, w_m) \in \mathbb{R}^m \bigm| u \in \mathbb{R} \big\}.$$

By hypothesis, ρ is defined by $k \leq n$ polynomials $p_1, \ldots, p_k \in \mathbb{R}[\boldsymbol{x}]$ and distinct indices $i_1, \ldots, i_k \in \{1, \ldots, n\}$ such that, for each $\boldsymbol{v} \in \mathbb{R}^n$,

$$\rho(\boldsymbol{v}) = \big\{ (v'_1, \ldots, v'_n) \in \mathbb{R}^n \bigm| v'_{i_1} = p_1(\boldsymbol{v}), \ldots, v'_{i_k} = p_k(\boldsymbol{v}) \big\}.$$

Then, for each term $\boldsymbol{x}^{\boldsymbol{\alpha}} \in \mathbb{R}_d[\boldsymbol{x}]$ where $\boldsymbol{\alpha} \neq \boldsymbol{0}$, we distinguish the following cases:

- Suppose there exists $j \in \{1, \ldots, n\}$ such that $\alpha_j > 0$ and $j \notin \{i_1, \ldots, i_k\}$. This means that ρ nondeterministically updates at least one of the relevant factors of the term $\boldsymbol{x}^{\boldsymbol{\alpha}}$. Thus, we conservatively approximate the overall effect of ρ on $\boldsymbol{x}^{\boldsymbol{\alpha}}$ using $\boldsymbol{x}^{\boldsymbol{\alpha}} \mapsto ?$, as if it was a nondeterministic assignment.
- Suppose now that, for each $j = 1, \ldots, n$, if $\alpha_j > 0$ then $j \in \{i_1, \ldots, i_k\}$, i.e., all the relevant factors of $\boldsymbol{x}^{\boldsymbol{\alpha}}$ are deterministically updated by ρ. Then:
  - if the polynomial $p_{\boldsymbol{\alpha}} := \prod\big\{ p_h^{\alpha_j}(\boldsymbol{x}) \bigm| j \in \{1, \ldots, n\}, \alpha_j > 0, j = i_h \big\}$ is such that $p_{\boldsymbol{\alpha}} \in \mathbb{R}_d[\boldsymbol{x}]$, we apply the affine map $\boldsymbol{x}^{\boldsymbol{\alpha}} \mapsto p_{\boldsymbol{\alpha}}$;
  - otherwise, since we cannot represent the effect of ρ on $\boldsymbol{x}^{\boldsymbol{\alpha}}$, we (again) conservatively overapproximate it as $\boldsymbol{x}^{\boldsymbol{\alpha}} \mapsto ?$.

Since ρ updates all terms simultaneously, these maps are ordered topologically according to the dependencies of terms (possibly adding temporary copies of some term variables, which are eliminated at the end).

Example 3. Consider the transitions of Example 1. For the transition τ_0 we have $\rho_0 \equiv (a', b', c') = (a, b, c/2)$. This update is linearized by composing the affine maps $ac \mapsto ac/2$, $bc \mapsto bc/2$, $c^2 \mapsto c^2/4$ and $c \mapsto c/2$, leading to $\rho_0^\star$ defined as

$$\big(a', b', c', ab', ac', bc', (a^2)', (b^2)', (c^2)'\big) = (a, b, c/2, ab, ac/2, bc/2, a^2, b^2, c^2/4).$$

Test for inclusion. The test for inclusion can be conservatively overapproximated by means of the test for inclusion for convex polyhedra.

Widening. Any widening for convex polyhedra, e.g., the standard widening [14] or the more sophisticated widenings proposed in [2,3], will serve the purpose of guaranteeing termination, with different trade-offs between efficiency and precision.

Example 4. For the transitions of Example 1, using the abstract semantics shown above, we obtain the invariant

$$\begin{aligned}\mathrm{reach}_{\ell_0}(S) \implies \big\{&(b+c)^2 \geq a+1, a \geq b^2, b \geq 0, c \geq 1,\\ &a^2 \geq 0, ab \geq 0, ac \geq 0, b^2 \geq bc, bc \geq b, (c-1)^2 \geq 0\big\}.\end{aligned}$$

Notice that all the constraints appearing on the second line are in fact redundant. Some of these, such as $a^2 \geq 0$ and $(c-1)^2 \geq 0$, are trivially redundant in themselves. Other ones are made redundant by the constraints appearing on the first line (for instance, $ab \geq 0$ is implied by $a \geq b^2$ and $b \geq 0$). This phenomenon is due to the interaction of the enrich_d procedure, which adds redundant constraints to polynomial cones, with the underlying linear inequalities inference rules, which are treating different terms as independent variables and, as a consequence, are only able to detect and remove some of the redundancies.

The two constraints $(b+c)^2 \geq a+1$ and $a \geq b^2$ in the invariant above are essential in a formal proof of the (partial) correctness of the program in Figure 1. Note that the computed invariant *assumes* that the integer division $c := c/2$ is correctly modeled by rational division. Such an assumption can be validated by other analyses, e.g., by using a domain of numerical powers [21], which could infer that c evaluates to a power of 2 at location ℓ_0. Since on termination $c = 1$ holds, the conjunction of these constraints implies $(b+1)^2 > a \geq b^2$.

4 Experimental Evaluation

The approach described in this paper has been implemented in a prototype analyzer that infers polynomial inequalities of degree not greater than $d = 2$. The prototype, which is based on the *Parma Polyhedra Library* (PPL) [4], first performs a rather standard *linear* relations analysis, then assumes the linear invariants so obtained for the analysis of (possibly) nonlinear invariants described in the previous sections. We have observed that this preliminary linear analysis improves the results in a significant way. In fact: (1) it ensures that we never obtain less information than is achievable with the linear analysis alone; (2) the availability of "trusted" linear invariants increases the precision of the nonlinear analysis considerably; and (3) the time spent in the linear analysis phase is usually recovered in the quadratic analysis phase. The prototype uses the sophisticate widening operator proposed in [2] enhanced with variations of the "widening up to" technique described in [17] and with the "widening with tokens" technique (a form of delayed widening application) described in [3].

Considering that, with the chosen degree bound $d = 2$, we are working on an ambient space that has a dimension which is quadratic in the number of variables of the transition system being analyzed, and considering that polyhedra operations have exponential worst-case complexity, some care has to be taken in order to analyze systems of realistic complexity. In our prototype, we exploit the capability of the PPL concerning the support of time-bounded computations. All polyhedra operations are subject to a timeout (5 seconds of CPU time in the experimentation we are about to report); when a timeout expires, the PPL abandons (without leaking memory) the current computation and gives control back to the analyzer. This situation is handled by the analyzer by using a less precise operation (such as replacing the precise convex polyhedral hull of two polyhedra P_1 and P_2 by the polyhedron obtained by removing, from a system

Table 1. A summary of the experimental results

					Linear analysis		Quadratic analysis	
Program name	Origin	n	$\lvert\mathcal{L}\rvert$	$\lvert\mathcal{T}\rvert$	CPU time	vs StInG	CPU time	Improves
array		4	5	6	0.2	$+\neq\neq\neq+$	79.8	✓
bakery	[34]	2	9	24	18.6	$=\cdots=$	0.2	✓
barber	FAST	8	1	12	18.7	$-$	2.7	✓
berkeley	FAST	4	1	3	0.0	$+$	0.1	✓
cars	StInG	7	1	2	18.5	$\neq$	45.9	✓
centralserver	FAST	12	1	8	5.4	$+$	193.4	
consistency	FAST	11	1	7	2.5	$=$	10.0	
consprodjava	FAST	16	1	14	325.6	$+$	601.9	
consprodjavaN	FAST	16	1	14	308.0	$+$	611.6	
cousot05vmcai	[9]	4	1	1	0.0	$=$	0.1	✓
csm	FAST	14	1	13	29.3	$=$	219.5	
dekker	FAST	22	1	22	458.4	$=$	1218.1	
dragon	FAST	5	1	12	0.5	$-$	1.4	✓
efm	FAST	6	1	5	0.1	$=$	0.3	
rfm05hscc	[28]	4	1	2	0.1	$\neq$	38.5	
firefly	FAST	4	1	8	0.1	$=$	0.2	✓
fms	FAST	22	1	20	893.2	$=$	2795.0	
freire	[16]	3	1	1	0.0	$-$	6.4	
futurbus	FAST	9	1	9	2.8	$+$	23.2	✓
heap	StInG	5	1	4	0.1	$\neq$	10.9	
illinois	FAST	4	1	9	0.1	$=$	0.3	✓
kanban	FAST	16	1	16	60.5	$=$	340.4	
lamport	FAST	11	1	9	3.1	$+$	13.4	
lifo	StInG	7	1	10	1.4	$+$	14.8	✓
lift	FAST	4	1	5	0.1	$=$	22.1	
mesi	FAST	4	1	4	0.0	$=$	0.1	✓
moesi	FAST	5	1	4	0.1	$-$	0.3	✓
multipoll	FAST	18	1	17	116.3	$=$	476.8	
peterson	FAST	14	1	12	17.6	$+$	88.5	
producer-consumer	FAST	5	1	3	0.1	$=$	15.5	
readwrit	FAST	13	1	9	7.7	$=$	2147.3	
rtp	FAST	9	1	12	2.6	$=$	8.9	
see-saw	StInG	2	1	4	0.0	$-$	5.3	
sqroot1	[15]	2	1	1	0.0	$=$	0.0	✓
sqroot2	[15]	3	1	8	0.0	$+$	15.6	✓
sqroot3	[15]	3	2	6	0.0	$==$	10.3	✓
sqroot4	[15]	4	2	6	10.3	$=+$	8.2	✓
sqroot5	[8]	4	1	1	0.0	$+$	6.1	✓
sqroot6	[18]	5	1	2	0.0	$=$	15.5	✓
swim-pool	StInG	9	1	6	1.5	$+$	46.5	
synapse	FAST	3	1	3	0.0	$+$	0.0	✓
ticket2i	FAST	6	1	6	0.3	$+$	5.8	
ticket3i	FAST	8	1	9	9.5	$+$	82.6	
train-beacon	StInG	3	4	12	0.1	$=--=$	20.5	
train-one-loc	StInG	3	1	6	0.0	$-$	0.4	
ttp	FAST	9	4	17	9.3	$++++$	126.9	

of constraints defining P_1, all constraints that are not satisfied by P_2) or by simplifying the involved polyhedra resorting to a domain of bounded differences. With this technique we are able to obtain results that are generally quite precise in reasonable time (note that the prototype was not coded with speed in mind).

We have run the prototype analyzer on a benchmark suite constituted by all the programs from the FAST suite [5] (http://www.lsv.ens-cachan.fr/fast/), programs taken from the StInG suite [29] (http://www.stanford.edu/ srirams/Software/sting.html), all square root algorithms in [15], programs from [9,18,28,34], and a program, `array`, written by the authors. From the StInG suite we have only omitted those programs with nondeterministic assignments where the *rvalue* is bounded by linear expressions (like $0 \geq x' \geq x + y$), since they do not fall into the programming model used here.

A summary of the experimental results is presented in Table 1. Besides the program name, its origin and the number of variables, locations and transitions (columns from 1 to 5, respectively), the table indicates: (1) the CPU time, in seconds, taken to compute our *linear* invariants (column 6) and how they compare with the ones computed by StInG (column 7: '+' means ours are better, '−' means ours are worse, '=' means they are equal, '≠' means they are not comparable); and (2) the time taken to generate *quadratic* invariants (column 8) and whether these invariants improve upon (that is, are not implied by) the linear ones, *taking into account both our linear invariants as well as those generated by StInG* (column 9: '✓' means we improve the precision). The measurements were performed on a PC with an Intel® Xeon$^{\text{TM}}$ CPU clocked at 1.80 GHz, equipped with 1 GB of RAM and running GNU/Linux. Notice that for about 80% of the locations, our linear invariants are at least as strong as the ones produced by StInG, and that, in fact, for one third ours are stronger. Most importantly, for about half of the programs, the obtained quadratic invariants improve the precision of the linear analysis.

5 Conclusion

We have presented a technique for generating invariant polynomial inequalities of bounded degree. The technique, which is based on the abstract interpretation framework, consists in overapproximating basic semi-algebraic sets by means of convex polyhedra, and can thus take advantage of all the work done in that field (e.g., refined widening operators, devices able to throttle the complexity of the analysis such as restricted classes of polyhedra, ways of partitioning the vector space and so forth). The application of our prototype implementation to a number of benchmark programs shows that the method can produce non-trivial and useful quadratic invariant inequalities in reasonable time, thus proving the feasibility of the automatic inference of nonlinear invariant inequalities (something that was previously unclear).

For future work, we want to generalize our definition of basic semi-algebraic transition system so as to capture a form of nondeterministic assignments where the *rvalue* is bounded by means of polynomial inequalities, rather than being

completely unknown. We would also like to increase the precision of the approach by incorporating, in the enrich$_d$ algorithm, other forms of inference, such as *relational arithmetic* [1,32]. This technique allows to infer constraints on the qualitative relationship of an expression to its arguments and can be expressed by a number of axiom schemata such as $(x > 0 \wedge y > 0) \implies \bigl(x \bowtie 1 \implies xy \bowtie y\bigr)$, which is valid for each $\bowtie \in \{=, \neq, \leq, <, \geq, >\}$. Finally, there is much room for improving the prototype implementation. To start with, we believe its performance can be greatly enhanced (there are a number of well-known techniques that we are not currently using); this may even bring us to the successful inference of cubic invariants for simple programs. The simplification of the analysis results is another natural candidate for this line of work.

Acknowledgments. The authors are grateful to Deepak Kapur and Alessandro Zaccagnini for their help and comments.

References

1. R. Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages.* PhD thesis, Dipartimento di Informatica, Università di Pisa, Pisa, Italy, March 1997.
2. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In *Proc. SAS 2003*, vol. 2694 of *LNCS*, pp. 337–354, San Diego.
3. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 2005. To appear.
4. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *Proc. SAS 2002*, vol. 2477 of *LNCS*, pp. 213–229, Madrid, Spain.
5. S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Fast Acceleration of Symbolic Transition systems. In *Proc. CAV 2003*, vol. 2725 of *LNCS*, pp. 118–121, Boulder, CO.
6. S. Bensalem, M. Bozga, J.-C. Fernandez, L. Ghirvu, and Y. Lakhnech. A transformational approach for generating non-linear invariants. In *Proc. SAS 2000*, vol. 1824 of *LNCS*, pp. 58–74, Santa Barbara, CA.
7. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. PLDI 2003*, pp. 196–207, San Diego, CA.
8. M. Colón. Approximating the algebraic relational semantics of imperative programs. In *Proc. SAS 2004*, vol. 3148 of *LNCS*, pp. 296–311, Verona, Italy.
9. P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *Proc. VMCAI 2005*, vol. 3385 of *LNCS*, pp. 1–24, Paris, France.
10. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. ISOP 1976*, pp. 106–130, Paris, France.
11. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL 1977*, pp. 238–252, New York.
12. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. POPL 1979*, pp. 269–282, New York.

13. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *Proc. ESOP 2005*, vol. 3444 of *LNCS*, pp. 21–30, Edinburgh, UK.
14. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL 1978*, pp. 84–96, Tucson, AR.
15. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
16. P. Freire. SQRT. Retrieved April 10, 2005, from `http://www.pedrofreire.com/sqrt`, 2002.
17. N. Halbwachs. Delay analysis in synchronous programs. In *Proc. CAV 1993*, vol. 697 of *LNCS*, pp. 333–346, Elounda, Greece.
18. P. Hsieh. How to calculate square roots. Retrieved April 10, 2005, from `http://www.azillionmonkeys.com/qed/sqroot.html`, 2004.
19. D. Kapur. Automatically generating loop invariants using quantifier elimination. In *Proc. ACA 2004*, Beaumont, Texas.
20. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.
21. I. Mastroeni. Algebraic power analysis by abstract interpretation. *Higher-Order and Symbolic Computation*, 17(4):297–345, 2004.
22. A. Miné. The octagon abstract domain. In *Proc. WCRE'01*, pp. 310–319, Stuttgart, Germany.
23. M. Müller-Olm and H. Seidl. Computing polynomial program invariants. *Information Processing Letters*, 91(5):233–244, 2004.
24. M. Müller-Olm and H. Seidl. A note on Karr's algorithm. In *Proc. ICALP 2004*, vol. 3142 of *LNCS*, pp. 1016–1028, Turku, Finland.
25. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Proc. POPL 2004*, pp. 330–341, Venice, Italy.
26. E. Rodríguez-Carbonell and D. Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In *Proc. SAS 2004*, vol. 3148 of *LNCS*, pp. 280–295, Verona, Italy.
27. E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *Proc. ISSAC 2004*, pp. 266–273, Santander.
28. M. Roozbehani, E. Feron, and A. Megrestki. Modeling, optimization and computation for software verification. In *Proc. HSCC 2005*, pp. 606–622, Zürich.
29. S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint-based linear-relations analysis. In *Proc. SAS 2004*, vol. 3148 of *LNCS*, pp. 53–68, Verona, Italy.
30. S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Proc. VMCAI 2005*, vol. 3385 of *LNCS*, pp. 25–41, Paris, France.
31. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *Proc. POPL 2004*, pp. 318–329, Venice, Italy.
32. R. Simmons. Commonsense arithmetic reasoning. In *Proc. AAAI 1986*, vol. 1, pp. 118–124, Philadelphia, PA.
33. J. Stoer and C. Witzgall. *Convexity and Optimization in Finite Dimensions I.* Springer-Verlag, Berlin, 1970.
34. A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In *Proc. TACAS 2001*, vol. 2031 of *LNCS*, pp. 113–127, Genova, Italy.

Inference of Well-Typings for Logic Programs with Application to Termination Analysis

Maurice Bruynooghe[1,⋆], John Gallagher[2,⋆⋆], and Wouter Van Humbeeck[1]

[1] Katholieke Universiteit Leuven, Department of Computer Science,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
Maurice.Bruynooghe@cs.kuleuven.ac.be

[2] Roskilde University, Computer Science, Building 42.1
DK-4000 Roskilde, Denmark
jpg@ruc.dk

Abstract. A method is developed to infer a polymorphic well-typing for a logic program. Our motivation is to improve the automation of termination analysis by deriving types from which norms can automatically be constructed. Previous work on type-based termination analysis used either types declared by the user, or automatically generated monomorphic types describing the success set of predicates. The latter types are less precise and result in weaker termination conditions than those obtained from declared types. Our type inference procedure involves solving set constraints generated from the program and derives a well-typing in contrast to a success-set approximation. Experiments so far show that our automatically inferred well-typings are close to the declared types and result in termination conditions that are as strong as those obtained with declared types. We describe the method, its implementation and experiments with termination analysis based on the inferred types.

1 Introduction and Motivation

For a long time, the selection of the right norm was a barrier to progress towards the full automation of termination analysis of logic programs. Recently, type-based norms have been introduced [23] as well as a technique to perform an analysis based on several norms [8]. There is evidence that the combination of both techniques solves in many cases the problem of norm selection [13,1]. However, most logic programs are untyped. Hence, obtaining type information is a new barrier to full automation. Systems for the automated inference of types do exist [7,24]. They derive monomorphic types that approximate the success-set of the program, and such inferred types are used to generate norms in a system for termination analysis [13]. Success types cannot in general be used directly by methods that require a well-typing [1]. In any case, inferred types obtained by

⋆ Work supported by FWO-Vlaanderen and by GOA/2003/08.

⋆⋆ Work supported in part by European Framework 5 Project ASAP (IST-2001-38059), and the IT-University of Copenhagen.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 35–51, 2005.
© Springer-Verlag Berlin Heidelberg 2005

current methods are often less precise than declared types, which are not necessarily over-approximations of the success set. The derived termination conditions are thus weaker than those obtained with declared types. The type inference described in this paper yields well-typings rather than success-set approximations and in all experiments so far yield types – and hence termination conditions – comparable to user-declared types.

We start by sketching an example of type-based termination analysis [1].

Example 1. Consider the `append/3` predicate and its abstraction according to the type signature `append(list(T),list(T),list(T))`. Each argument is abstracted by the type-based `list(T)` norm that abstracts a term by the number of subterms of type `list(T)` and the type-based `T` norm that abstracts a term by the number of subterms of type `T` (subscripts l and e for abstracted variables).

```
append([],L,L).
append([X|Xs],Ys,[X|Zs]):- append(Xs,Ys,Zs).
append(1,0, 1+L_l,L_e, 1+L_l,L_e).
append(1+1+Xs_l,1+X_e+Xs_e, 1+Ys_l,Ys_e, 1+1+Zs_l,1+X_e+Zs_e):-
  append(1+Xs_l,Xs_e, 1+Ys_l,Ys_e, 1+Zs_l,Zs_e).
```

This suffices to infer that a call to append/3 terminates if it is `list(T)`-rigid[1] in either the first or the last argument. A goal independent type inference [7,24] infers the type append(list(any),any,any), giving rise to the abstract program:

```
append(1,0, 1+L_a, 1+L_a).
append(1+1+Xs_l,1+X_a+Xs_a, 1+Ys_a, 1+1+X_a+1+Zs_a):-
  append(1+Xs_l,Xs_a, 1+Ys_a, 1+Zs_a).
```

The subscripts l and a of abstracted variables correspond to respectively the `list(any)` and `any`-norm; a term of type `any` has only subterms of type `any`, so the second and third argument have only an `any`-abstraction. The termination condition for the third argument is weaker than with the declared type as it requires `any`-rigidity and this corresponds to groundness.

In this paper, the signature `append(a1(T),a2(T),a2(T))` is inferred, with the types defined as $\mathtt{a_1(T)} \longrightarrow$ `[]; [T|a1(T)]` and $\mathtt{a_2(T)} \longrightarrow$ `[T|a2(T)]`. The type $\mathtt{a_1(T)}$ is equivalent to `list(T)`; the type $\mathtt{a_2(T)}$ may look odd as it lacks a "base case" but it gives a well-typing. Calls such as `append([a],[b|X],Y)` are well-typed, and give rise to well-typed calls in their computations. In short "well-typed programs do not go wrong" even with such peculiar types. Now, the abstracted program is:

```
append(1,0, 1+L_a2,L_T, 1+L_a2,L_T).
append(1+1+Xs_a1,1+X_T+Xs_T, 1+Ys_a2,Ys_T, 1+1+Zs_a2,1+X_T+Zs_T):-
  append(1+Xs_a1,Xs_T, 1+Ys_a2,Ys_T, 1+Zs_a2,Zs_T).
```

Hence calls terminate when $\mathtt{a_1}$-rigid in the first or $\mathtt{a_2}$-rigid in the third argument.

[1] Rigid: all instances have the same size under the norm.

As the next example shows, a call from outside can extend the type of a predicate.

Example 2. The naive reverse procedure is given by the clauses

```
rev([],[]).
rev([X|Xs],Zs) :- rev(Xs,Ys),append(Ys,[X],Zs]).
```

together with the clauses for `append`. The inferred signatures and types are

```
t1(T) --> [T|t1(T)]; []      rev(t2(T),t1(T)).
t2(T) --> [T|t2(T)]; []      app(t1(T),t1(T),t1(T))
```

Note that the two types denote the same set of terms. The analysis derives two distinct types because the cons-functors of both do not interact with each other.

Example 3. A program to transpose a matrix represented as a list of rows [1]:

```
transpose(A,B) :- transpose_aux(A,[],B).
transpose_aux([], W, W).
transpose_aux([R|Rs],Z,[C|Cs]) :-
      row2col(R,[C|Cs],Cls1,[],Acc), transpose_aux(Rs,Acc,Cls1).
row2col([],[],[],A,A).
row2col([X|Xs],[[X|Ys]|Cols],[Ys|Cols1],B,C) :-
      row2col(Xs,Cols,Cols1,[[]|B],C).
```

The inferred signature and type definitions are as follows:

```
t1(T) ⟶ []; [t4(T)|t1(T)]      transpose(t1(T),t2(T))
t2(T) ⟶ []; [t3(T)|t2(T)]      transpose_aux(t1(T),t2(T),t2(T))
t3(T) ⟶ []; [T|t3(T)]          row2col(t3(T),t2(T),t2(T),t2(T),t2(T))
t4(T) ⟶ []; [T|t4(T)]
```

The types `t3(T)` and `t4(T)` are equivalent and denote a row of elements `T`. Also `t1(T)` and `t2(T)` are equivalent; they denote a list of rows of `T`. These types are equivalent to what a programmer would declare: the first argument of `row2col/5` is a row and all others are lists of rows. Types inferred by over-approximation of success sets using current techniques, even when using a goal-directed analysis with the goal `transpose(X,Y)` are less accurate.

We define the basic notions of types and set constraints in Section 2; we present the type inference procedure in Section 3. The implementation, complexity and some experiments in both type inference and the use of the types in termination analysis are described in Section 4. An extension for obtaining more polymorphism is given in Section 5. Finally we discuss related work in Section 6 and future research in Section 7.

2 Preliminaries

2.1 Types

For type definitions, we adopt the syntax of Mercury [19]. *Type expressions (types)*, elements of $\mathcal{T}$, are constructed from an infinite set of type variables (parameters) $\mathcal{V}_\mathcal{T}$ and an alphabet of ranked type symbols $\Sigma_\mathcal{T}$; these are disjoint from the set of variables V and alphabet of functors Σ used to construct terms. Variable free types are called monomorphic; the others polymorphic. Type substitutions of the form $\{T_1/\tau_1, \ldots, T_n/\tau_n\}$ with the T_i parameters and the τ_i types define mappings from types to types by the simultaneous replacement of the parameters T_i by the corresponding types τ_i.

Definition 1 (Type definition). *A* type rule *for a type symbol* $h/n \in \Sigma_\mathcal{T}$ *is of the form* $h(\bar{T}) \longrightarrow f_1(\bar{\tau}_1); \ldots; f_k(\bar{\tau}_k);\ \ (k \geq 1)$ *where* $\bar{T}$ *is a* n*-tuple of distinct type variables,* $f_1, \ldots, f_k$ *are distinct function symbols from* Σ, $\bar{\tau}_i$ $(1 \leq i \leq k)$ *are tuples of corresponding arity from* $\mathcal{T}$*, and type variables in the right hand side, if any, are from* $\bar{T}$[2]*. A* type definition *is a finite set of type rules where no two rules contain the same type symbol on the left hand side, and there is a rule for each type symbol occurring in the type rules.*

A *predicate signature* is of the form $p(\bar{\tau})$ and declares a type τ_i for each argument of the predicate p/n. The mapping $\bar{\tau}_i \rightarrow h(\bar{T})$ can be considered the type signature of the function symbol f_i. As in Mercury [19], a function symbol can occur in several type rules, hence can have several type signatures.

A *typed* logic program consists of a logic program, a type definition and a predicate signature for each predicate of the program. Given a typed logic program, a type checker can verify whether the program is well-typed, i.e., that the types of the actual parameters passed to a predicate are an instance of the predicate's type signature. To formalize the well-typing, we first inductively define the well-typing of a term.

Definition 2. *A* variable typing *is a mapping from variables to types. A term* t *has type* $h(\bar{\tau})$ *(notation* $t : h(\bar{\tau})$*) under a variable typing* μ *iff either* t *is a variable* X *and* $\mu(X) = h(\bar{\tau})$ *or* t *is of the form* $f(t_1, \ldots, t_n)$*, the type rule for* $h(\bar{T})$ *has an alternative* $f(\tau_1, \ldots, \tau_n)$ *and, for all* i, t_i *has type* $\tau_i\{\bar{T}/\bar{\tau}\}$*.*

Definition 3 (Well-typing). *A typed program* P *has a well-typing if each clause* $p(t_1, \ldots, t_n) \leftarrow B_1, \ldots, B_m \in P$ *has a variable typing* μ *that satisfies:*

1. *Let* $p(\tau_1, \ldots, \tau_n)$ *be the predicate signature of* p/n*. Then* t_i *has the type* τ_i *under the variable typing* μ $(1 \leq i \leq n)$*.*
2. *For* $1 \leq j \leq m$*, let* $B_j = q(s_1, \ldots, s_l)$ *and* $q(\tau_1, \ldots, \tau_l)$ *be the predicate signature of* q/l*. Then there is a type substitution* θ *such that, for all* k, s_k *has type* $\tau_k\theta$ *under the variable typing* μ*.*

[2] The last condition is known as *transparency* and is necessary to ensure that well-typed programs cannot go wrong [17,10].

Example 4. Given a type definition `list(T)` $\longrightarrow$ `[]; [T | list(T)]` the signature `append(list(T),list(T),list(T))` gives a well-typing of the program of Example 1. The variable typing of the first clause is {`L/list(T)`} and that of the second clause is {`X/T,Xs/list(T),Ys/list(T),Zs/list(T)`}.

To establish the connection with set constraints (Section 2.2) we formalize the denotation of a type. Let $D : \mathcal{V}_\mathcal{T} \to 2^{\mathsf{Term}_\Sigma}$ be a mapping from parameters to sets of ground terms. Let $h(\bar{\tau})$ be defined by the type rule $h(\bar{T}) \longrightarrow f_1(\bar{\tau}_1); \ldots; f_k(\bar{\tau}_k)$. Using e_j to denote the jth element in a sequence $\bar{e}$, the denotation of $h(\bar{\tau})$ with respect to D, written $\mathsf{Den}_D(h(\bar{\tau}))$ is inductively defined as:

1. For all $T \in \mathcal{V}_\mathcal{T}$, $\mathsf{Den}_D(T) = D(T)$.
2. $\mathsf{Den}_D(h(\bar{\tau})) = \{f_i(\bar{s}) \mid 1 \le i \le k, s_j \in \mathsf{Den}_D(\tau_{i_j}\{\bar{T}/\bar{\tau}\}) \text{ for all } j\}$.

Proposition 1. *Let $t[\bar{X}]$ denote a term with variables $\bar{X}$; μ a variable typing and $D : \mathcal{V}_\mathcal{T} \to 2^{\mathsf{Term}_\Sigma}$ a mapping from type variables to sets of ground terms. Then $t[\bar{X}]$ has type $h(\bar{\tau})$ under μ iff $\mathsf{Den}_D(h(\bar{\tau})) \supseteq \{t[\bar{X}]\{\bar{X}/\bar{s}\} \mid s_i \in \mathsf{Den}_D(\mu(X_i))\}$.*

2.2 Set Constraints for Well-Typings

Set Constraints and Their Solutions. Set expressions are terms constructed from an infinite set of set variables $\mathcal{V}_\mathcal{S}$ and the same alphabet of functors Σ as used for constructing terms. Given a mapping $V : \mathcal{V}_\mathcal{S} \to 2^{\mathsf{Term}_\Sigma}$ from set variables to sets of ground terms, one can inductively define the denotation for set expressions e with respect to V, written $\mathsf{Den}_V(e)$, as follows:

1. For all $s \in \mathcal{V}_\mathcal{S}$, $\mathsf{Den}_V(s) = V(s)$.
2. $\mathsf{Den}_V(f(e_1, \ldots, e_n)) = \{f(s_1, \ldots, s_n) \mid s_i \in \mathsf{Den}_V(e_i), 1 \le i \le n\}$.

The set constraints that we consider are of two kinds, namely $t_1 = t_2$ where $t_1, t_2 \in \mathcal{V}_\mathcal{S}$ and $t_1 \supseteq f(e_1, \ldots, e_n)$ where $t_1 \in \mathcal{V}_\mathcal{S}$ and $f(e_1, \ldots, e_n)$ is a set expression. We call set constraints of the first kind *equality* constraints and those of the second kind *containment* constraints.

Let $\mathcal{S}$ be a set of set constraints (or *constraint system*). A *solution* for $\mathcal{S}$ is any mapping $S : \mathcal{V}_\mathcal{S} \to 2^{\mathsf{Term}_\Sigma}$ such that for each constraint the following holds.

1. For all $t_1 = t_2 \in S$, $\mathsf{Den}_S(t_1) = \mathsf{Den}_S(t_2)$.
2. For all $t_1 \supseteq f(e_1, \ldots, e_n) \in S$, $\mathsf{Den}_S(t_1) \supseteq \mathsf{Den}_S(f(e_1, \ldots, e_n))$.

Solved Form and Normal Form. A constraint system $\mathcal{S}$ is in *solved form* if, for each equality constraint $t_1 = t_2$, t_1 has no other occurrences in $\mathcal{S}$.

Given a constraint system, one can derive an equivalent solved form by repeatedly taking an equality constraint $t_1 = t_2$ where t_1 has other occurrences and substituting t_1 by t_2 (or alternatively, replacing the equation by $t_2 = t_1$ and substituting t_2 by t_1) throughout the other constraints. Any resulting equalities $t = t$ are removed. As each such step reduces the number of set variables on the

left hand side of an equality with other occurrences, and no new variables are introduced, the process terminates and yields a solved form.

Let $\mathcal{S}$ be a constraint system in solved form, and let $t \in \mathcal{V}_\mathcal{S}$ be a set variable. Then t is *constrained* in $\mathcal{S}$ if t appears on the left hand side of a constraint, otherwise t is *unconstrained* in $\mathcal{S}$. Note that a constrained set variable in a solved form occurs either in the left hand side of one equality constraint or in the left hand side of one or more containment constraints. In constructing a solution for a constraint system $\mathcal{S}$ in solved form, one can freely choose a denotation for its unconstrained variables. We denote a solution for the unconstrained variables in $\mathcal{S}$ by U. Denote by $S[U]$ any solution of $\mathcal{S}$ that extends U.

Definition 4 (Minimal solution). *A solution $S[U]$ of $\mathcal{S}$ is* minimal *with respect to U iff for each solution $S'[U]$, it holds that for all set variables s, $S[U](s) \subseteq S'[U](s)$.*

We often omit U when it is not relevant and denote a solution by S.

Proposition 2. *Let S be a minimal solution of a constraint system $\mathcal{S}$ in solved form and t a set variable constrained by containment constraints. $f(\bar{s}) \in \mathsf{Den}_S(t)$ iff there is a containment constraint $t \supseteq f(\bar{e})$ such that $f(\bar{s}) \in \mathsf{Den}_S(f(\bar{e}))$.*

Definition 5 (Normal form). *A constraint system is in* normal form *if it is in solved form and additionally the following conditions are satisfied.*

1. *It does not contain two distinct containment constraints $t \supseteq f(e_1, \ldots, e_n)$ and $t \supseteq f(e'_1, \ldots, e'_n)$.*
2. *All e_i in containment constraints $t \supseteq f(e_1, \ldots, e_n)$ are set variables.*

Note that 1 corresponds to the requirement that functions symbols are distinct in the right hand side of a type rule. A constraint system $\mathcal{S}$ can be normalised by applying the following operations until a fixpoint is reached.

1. If $\mathcal{S}$ contains $t \supseteq f(e_1, \ldots, e_n)$ and $t \supseteq f(e'_1, \ldots, e'_n)$ with $e_1, \ldots, e_n, e'_1, \ldots, e'_n$ set variables then replace the latter by the constraints $e_1 = e'_1, \ldots, e_n = e'_n$.
2. If $\mathcal{S}$ contains $t \supseteq f(e_1, \ldots, e_j, \ldots, e_n)$ where e_j is not a set variable, then replace it by $t \supseteq f(e_1, \ldots, s, \ldots, e_n)$ and $s \supseteq e_j$ with s a fresh set variable.
3. Apply the rules for deriving a solved form.

Proposition 3. *The reduction to normal form terminates. Moreover, if $\mathcal{S}$ is a constraint system and $\mathcal{S}'$ is the normal form obtained by applying the procedure above, then every solution of $\mathcal{S}'$ is also a solution of $\mathcal{S}$.*

Given a normalised constraint system with a set variable t, we define *type*(t) as a type that has the same denotation as the minimal solution of t as follows.

First, define a directed graph with the set variables as nodes. For each constraint $t \supseteq f(s_1, \ldots s_n)$ add, for all i, the arc (t, s_i); for each constraint $t = s$

add the arc (t, s). Note that unconstrained variables have no out-arcs. For each constrained set variable t, define $params(t)$ to be the set of unconstrained variables reachable from t in the graph. For each unconstrained variable s define a unique type parameter T_s. For each variable t constrained by containment constraints, define a unique type symbol τ_t/n where $n = |params(t)|$.

Now, for each set variable t define $type(t)$ as T_t if t is unconstrained, as $type(s)$ if t is constrained by an equality constraint $t = s$, and as $\tau_t(T_1, \ldots, T_n)$ if t is constrained by containment constraints where $T_1, \ldots, T_n$ are the type parameters corresponding to $params(t)$ (enumerated in some order). To construct the type rules, let t be a constrained variable, and $t \supseteq f_1(\bar{t}_1), \ldots, t \supseteq f_m(\bar{t}_m)$ the containment constraints having t on the left. Then construct a type rule $type(t) \longrightarrow f_1(\bar{\tau}_1); \ldots; f_m(\bar{\tau}_m)$ where $\bar{\tau}_1, \ldots, \bar{\tau}_m$ are obtained from $\bar{t}_1, \ldots, \bar{t}_m$ by substituting each set variable $t_{i,j}$ by $type(t_{i,j})$.

Example 5. Consider the set variables $\mathtt{a}_1$, $\mathtt{a}_2$, $\mathtt{a}_3$ and the solved form $\mathtt{a}_1 \supseteq \mathtt{[]}$, $\mathtt{a}_1 \supseteq \mathtt{[x|a_1]}$, $\mathtt{a}_3 \supseteq \mathtt{[x|a_3]}$, $\mathtt{a}_2 = \mathtt{a}_3$.

The associated directed graph is $\{(\mathtt{a}_1, \mathtt{x}), (\mathtt{a}_1, \mathtt{a}_1), (\mathtt{a}_2, \mathtt{a}_3), (\mathtt{a}_3, \mathtt{x}), (\mathtt{a}_3, \mathtt{a}_3)\}$ The set variable $\mathtt{x}$ is unconstrained; let $type(\mathtt{x}) = \mathtt{X}$. We have $params(\mathtt{a}_1) = \{\mathtt{x}\}$ and $params(\mathtt{a}_3) = \{\mathtt{x}\}$. We use $\tau_{\mathtt{a}_i} = \mathtt{a}_i$, so $type(\mathtt{a}_1) = \mathtt{a}_1(\mathtt{X})$ and $type(\mathtt{a}_3) = \mathtt{a}_3(\mathtt{X})$. Hence the derived type rules are $\mathtt{a}_3(\mathtt{X}) \longrightarrow \mathtt{[X|\ a_3(X)]}$ and $\mathtt{a}_1(\mathtt{X}) \longrightarrow \mathtt{[];\ [X\ |\ a_1(X)]}$. Finally, $type(\mathtt{a}_2) = \mathtt{a}_3(\mathtt{X})$ because $\mathtt{a}_2 = \mathtt{a}_3$.

From Proposition 2 and the way the types are derived the following proposition follows immediately.

Proposition 4. *Let $\mathcal{S}$ be a constraint system in normal form and let $S[U]$ be a minimal solution. Let $type(s)$ be as defined above and let ρ denote the type definition derived from $\mathcal{S}$. For all $u \in domain(U)$ define $D(type(u)) = U(u)$. Then, for each set variable s it holds that $\mathsf{Den}_D(type(s)) = \mathsf{Den}_S[U](s)$.*

3 Inference of a Well-Typing

The purpose of type inferencing is to derive a typed program, that is, a type definition and a set of predicate signatures such that the program is well-typed. Whereas well-typing allows the type of a call to be an instance of the declared type, we will derive types such that they are equal. In Section 5 we outline a method for deriving truly polymorphic well-typings. Here, the approach is to associate a set variable with each type in the signatures of the predicates and one with each variable in the program code and to formulate a constraint system whose solution denotes a well typing. Then the constraint system is reduced to normal form. According to Proposition 3, its solutions are solutions of the original system, hence well-typings. From the normal form, the type definition is extracted as described in Section 2.2 and the predicate signatures are obtained by taking the types $type(s)$ of the corresponding set variables.

3.1 Generation of Constraints

Let P be a program. We introduce fresh set variables $p_1, \ldots, p_n$ for each predicate p/n of P and a fresh set variable t_x for each variable x of P[3]. In concrete examples we reuse the program variables as set variables in the constraints (that is, $t_x = x$), since there can be no confusion between them. The constraint system for a program is the union of the constraint systems generated for each atom in the program. The constraints generated from an atom $p(u_1, \ldots, u_n)$ are:

$$\{p_j \supseteq u_j \mid \text{if } u_j \text{ is not a variable}\} \cup \{p_j = u_j \mid \text{if } u_j \text{ is a variable}\}$$

Example 6. Consider the `append/3` program of Example 1. Using the set variables `ap1, ap2` and `ap3` for the `append/3` predicate, we obtain:

- From `append([],L,L)`: `ap1` $\supseteq$ `[]`, `ap2 = L`, `ap3 = L`.
- From `append([X|Xs],Ys,[X|Zs])`: `ap1`$\supseteq$`[X|Xs]`, `ap2 = Ys`, `ap3`$\supseteq$`[X|Zs]`.
- From `append(Xs,Ys,Zs)`: `ap1 = Xs`, `ap2 = Ys`, `ap3 = Zs`.

A normal form of this system consists of the constraints

```
ap1 ⊇ []     ap1 ⊇ [X|ap1]   ap3 ⊇ [X|ap3]
Ys = ap3     L = ap3         Xs = ap1        ap2 = ap3   Zs = ap3
```

As shown in Example 5, we obtain the following types and signature:

```
ap1(X) ⟶ []; [X | ap1(X)]            append(ap1(X),ap3(X),ap3(X))
ap3(X) ⟶ [X | ap3(X)]
```

While the type of the first argument is isomorphic to the `list(T)` type, that of the second and third argument is not as the `[]` alternative is not included. Interestingly, this type is accepted by Mercury [19]. It is only when `append/3` is called from elsewhere in the program as e.g. in the `rev` program of Example 2 that our type inference extends the type `ap3(X)` with a base case. The type inference on the `rev` program still results in two distinct (although equivalent) types. Although we are used to a signature `append(list(T),list(T),list(T))`, nothing in the code of `append/3` imposes this; `append(list(T),mylist(T),mylist(T))` where `mylist(T)` $\longrightarrow$ `mynil; [X | mylist(X)]` is an equally good signature. In fact, unless there is a call that imposes a base case, the choice of the base case is open, so one can argue that `ap3(X)`, a type without a base case is the most general and the most natural one.

Theorem 1. *The type signatures and the type rules derived from the normal form of the constraints generated from a program P are a well-typing for P.*

The proof follows immediately from Propositions 1, 3 and 4 (see [3]).

[3] We assume program clauses do not share variables and predicates p/n and p/m with $n \neq m$ do not occur.

4 Implementation and Experiments

The algorithm for type inference system consists of four main stages: (i) generation of the constraints from the program text, (ii) realisation of a solved form, (iii) normalisation and (iv) generation of the parameterised type definitions. Of these, normalisation is the only stage whose implementation requires careful consideration in order to be able to apply the system to larger programs.

Constraint Generation. One constraint is generated for each argument of each atom (see Section 3.1). This is achieved in a single pass over the program.

Solved Form. Constraint generation implies that the number of constraints is linear in the size of the program. Collecting the set of all the equalities, we compute the set of equivalence classes such that all members of an equivalence class are equal to each other. This can be done in time linear in the number of equality constraints. An element of each class is selected; denote by $rep(s)$ the selected element of s's class. The constraints of the solved form then consist of (i) the set of equalities $\{s = rep(s) \mid s \text{ is different from } rep(s)\}$ and (ii) the containment constraints with each variable s replaced by $rep(s)$. Given a suitable representation of the equivalence classes (see the discussion on *union-find* below) the substitution can be done in time proportional to the number of containment constraints. The resulting system is in solved form. Thus reduction to solved form can be achieved in linear time (with respect to the size of the program).

Normal Form. Normalisation is achieved starting from the containment constraints of the solved form. As described in Section 2.2, normalisation causes new constraints to be added, which can destroy solved form.

We focus on the removal of non-normal constraints $t \supseteq f(\bar{s}_1)$, $t \supseteq f(\bar{s}_2)$; the other case of non-normal constraints is trivial and can be removed in one pass. The algorithm for producing normal form is as follows, in outline.

```
Initialise equivalence classes, one class per variable;
while (not in normal form) {
  Pick a pair of constraints t1 ⊇ f(...) and t2 ⊇ f(...),
    where t1 and t2 are in the same equivalence class;
  Generate the appropriate constraints to remove the violation;
  Adjust equivalence classes using the generated equalities;
endwhile
```

The adjustment of the equivalence classes is essentially merging; when a constraint $s = t$ is generated we merge the equivalence classes of which s and t respectively are members. All the containment constraints whose left-hand-sides are in the same equivalence class are stored with the representative element of that class. The management of the equivalence classes uses the well-known *union-find* algorithms [21], so that the adjustment of the equivalence classes, and the location of the representative for a given class, can be done in close to constant time.

Thus the time taken to normalise is roughly linear in the number of constraints generated during normalisation. This is not directly determined by the

size of the program, since it depends on the distribution of variables in the program, the number of clauses for each predicate, and so on. However for typical programs the number of generated constraints is roughly proportional to the size of the program.

Conversion to parametrised type definitions. The procedure for finding the parameters involves constructing the dependency graph and finding the reachable unconstrained variables from each constrained variable, as described in Section 3. The time required for reachability computation is proportional to the number of normalised constraints, for each constrained variable.

In summary, each stage can be achieved efficiently in time roughly proportional to the size of the program. In our Prolog implementation, the elements of the equivalence classes in the union-find algorithm are stored in a balanced tree, thus giving logarithmic-time rather than constant-time execution of the *find* operation. Our experiments confirm that the running time of the type inference is roughly $O(n.log(n))$ where n is the size of the program.

4.1 Inference Experiments

We applied the procedure to a range of programs from the termination analysis literature as well as many other programs (including the implementation of the procedure itself). The procedure shows reasonable scalability: space does not permit a detailed table of statistics so we quote a few timings to give an impression. The largest program we attempted is the Aquarius compiler benchmark (4,192 clauses, 19,976 generated constraints, 18,168 normalisation constraints) for which type inference takes approximately 100 seconds on a Macintosh Powerbook G4. The Chat parser (515 clauses, 2,010 generated constraints, 1,613 normalisation constraints) requires 4.5 seconds. Programs of 100 clauses or less are analysed in fractions of a second. The software runs in Ciao or SICStus Prolog and can be downloaded from `http://www.ruc.dk/~jpg/Software/`. A sample of derived types can be found in [3].

4.2 Termination Analysis Experiments

We took a set of 45 small programs from [1] (most of them in turn are from the experiments in [13]) which included declared types. We compared the termination conditions obtained from the inferred types with those obtained from the declared types. We did so using the TerminWeb analyser [20]. On all examples, the termination conditions were equivalent.

In a second experiment, we inferred regular types [6] that approximate the success set of the program and used them for type-based termination analysis. Regular types are not always well-typings. As TerminWeb expects well-typings, we used the cTI termination analyser [13] for this experiment. The system is weaker than TerminWeb and cannot prove termination for 4 of the programs. For 3 programs, termination conditions are obtained with the well-typing but not with the regular types. For 14 programs, the termination conditions are

equivalent. For the remaining 24 programs, the well-typing results in stronger termination conditions. Typically, using the regular types, some argument is required to be ground while rigidity of some type constituent suffices when using the well-typing.

It is interesting to compare the inferred types with the declared types. For 27 of the 45 programs the inferred type is equivalent to the declared types in the sense that there is a simple renaming of type symbols that maps the inferred types to the declared types. The reverse mapping is not always possible, because sometimes distinct types are inferred that are a renaming of each other (and hence of a single declared type). Moreover, in most remaining cases one can say that the inferred type is more precise in the sense that the type allows fewer cases. Typically, a base case is missing as in the type `ap3(X)` $\longrightarrow$ `[X | ap3(X)]` of the third argument of `append`. For two programs, `der` and `parse`, the analysis distinguishes somewhere two types whereas the declared type has a single type that is the union of both. For the program `minimum` shown in Example 9 of Section 5 there is a more substantial difference. The declared type signature is `minimum(t(X),X)` with type rule `t(X)` $\longrightarrow$ `void; tree(X,t(X),t(X))`. The code in question does not access the right branch of the tree, hence there is no reason to infer it is a tree; the type inference derives the signature `minimum(t1(X,Y),X)` with `t1(X,Y)` $\longrightarrow$ `void; tree(X,t1(X,Y),Y)`. This difference is irrelevant when analysing termination. In this case one can observe that the declared type is an instance of the inferred type, since the denotations of `t(X)` and `t1(X,t(X))` are the same.

This experiment suggests that the types we infer are comparable to those one would declare. Often they are identical, and in the remaining cases, the most frequent situation is that the solved form that corresponds to the declared types is an extension of the solved form derived by our analysis.

5 Towards Inference of a Polymorphic Well-Typing

So far we derive a single signature for a predicate `p` that is valid for all its occurrences. While we do derive parametric types, our types are not truly polymorphic, because we insist that the type of a call is identical to the signature of the predicate rather than being an instance of it. When using the types for type-based termination analysis, polymorphic types are potentially more useful since the norms are more simple and more reuse of results is feasible [2,13]. We develop an extension where the type of calls can be different instances of the predicates signatures. First we illustrate the difficulty of achieving this.

Example 7. Consider the artificial program P consisting of the clause `p :- append([a],[b],M), append([M],[M],R).` together with P_{app}, the definition of `append`. The relevant part of the normal form of the constraint system generated from P_{app} and the extracted well-typing are respectively

```
ap1 ⊇ []      ap1 ⊇ [X|ap1]              ap3 ⊇ [X|ap3]      ap2 = ap3
ap1(X) ⟶ []; [X | ap1(X)]                append(ap1(X),ap3(X),ap3(X))
ap3(X) ⟶ [X | ap3(X)]
```

The extra constraints on `append` coming from the `p` clause are `ap1` $\supseteq$ `[a]`, `ap2` $\supseteq$ `[b]`, `ap3` $=$ `M`, `ap1` $\supseteq$ `[M]`, `ap2` $\supseteq$ `[M]`, and `ap3` $=$ `R`. The constraints on `ap1` and `ap2` give rise to `M = X`, `X` $\supseteq$ `a`, `X` $\supseteq$ `b` and `ap3` $\supseteq$ `[]`. Finally, `ap3` $=$ `M` enforces the same type for `X` and `ap3`; hence we obtain the signature `append(ap`$_1$`,ap`$_3$`,ap`$_3$`)` with the types `ap`$_1$ $\longrightarrow$ `[]; [ap`$_3$ $|$ `ap`$_1$`]` and `ap`$_3$ $\longrightarrow$ `[]; a; b; [ap`$_3$ $|$ `ap`$_3$`]`.

Note that one cannot obtain types equivalent to the latter signature by instantiating the type parameter of the former. Moreover, we obtain an imprecise type for `ap`$_3$ that includes `a` and `b` as alternatives because the constraints imply that all calls to `append` have the same type.

Procedure for Deriving Polymorphic Types. We first introduce some concepts and notations. A predicate p *depends directly* on a predicate q when q occurs in the right hand side of a clause with p in the head. A set variable s *depends directly* on a set variable t when t occurs in the right hand side of a constraint with s in the left hand side. In both cases, the *depends* relation is the transitive closure of the directly depends relation. With P_p, we denote the part of a program defining predicate p and the predicates p depends on. With $\mathcal{S}_P$, we denote the constraint system generated by program P. With $\mathcal{S}^{\bar{p}}$, we denote the part of the normal form of $\mathcal{S}$ that contains all constraints with on the left hand side either one of the p_i or a set variable on which one of the p_i depends, i.e., the part of the normal form needed to construct the complete type definitions of the types $type(p_i)$. With $\rho_i(\mathcal{S})$ we denote a renaming of $\mathcal{S}$ where each set variable s is replaced by s^i. Finally, when using $s_{\equiv}$ in the context of $\mathcal{S}$, we mean either s itself or a t such that $s = t$ belongs to the normal form of $\mathcal{S}$.

Now consider the partitioning of a program in two parts P and Q such that if P has a clause with head p, then it has all clauses with as head either p or predicates on which p depends[4]. Our goal is to derive a well-typing for all predicates such that the variable typing in Q of calls to P are instances of the (polymorphic) signatures of the predicates in P. As shown in Example 7, this is not straightforward to achieve. For each call $p(\bar{t})$ in Q to a predicate in P, we assume that the function $id(p(\bar{t}))$ returns an index that is unique for the call. From P we generate the constraint system $\mathcal{S}_P$ as described in Section 3.1. When generating $\mathcal{S}_Q$, calls $p(\bar{t})$ to predicates in P are treated differently. Instead of the constraints $p_j \ rel \ t_j$ (with $rel \in \{=, \supseteq\}$), we generate $\rho_{id(p(\bar{t}))}(p_j) \ rel \ t_j$ (the left hand side is renamed); moreover we add to $\mathcal{S}_Q$ the constraint system $\rho_{id(p(\bar{t}))}(\mathcal{S}_P^{\bar{p}})$, a renaming of the constraints relevant for $type(p_j)$ (for all j). Creating a different instance for each call ensures that each call can have a distinct well-typing. Note that $\mathcal{S}_P$ and $\mathcal{S}_Q$ do not share any set variables.

Next, the following operations are exhaustively applied on (the normal form of) $\mathcal{S}_P$ and $\mathcal{S}_Q$.

1. Let q be a set variable from S_P with $type(q)$ not a type parameter. If, for some i, ${q^i}_{\equiv} \supseteq f(\bar{t}) \in S_Q$ and there is no $\bar{s}$ such that $q_{\equiv} \supseteq f(\bar{s}) \in S_P$ (i.e., q contributes to the type signature of one or more predicates in P and $type(q)$ has no case for functor f while $type(q^i)$ of the signature of the call with

[4] More generally, one could consider a partition of strongly connected components.

identifier i does) then add $q \supseteq f(\bar{r})$ to $\mathcal{S}_P$ with $\bar{r}$ new set variables[5] and, for all j such that q^j exists in Q, add $\rho_j(q \supseteq f(\bar{r}))$ to $\mathcal{S}_Q$ (all copies in Q are updated).

2. Let s and t be different set variables in S_P such that s depends on t or t on s. If, for some i, $s^i_{\equiv} = t^i_{\equiv} \in \mathcal{S}_Q$ and $s_{\equiv} = t_{\equiv} \notin \mathcal{S}_P$, then add $s = t$ to $\mathcal{S}_P$ and, for all $j \neq i$ such that s^j exists in Q, add $\rho_j(s = t)$ to $\mathcal{S}_Q$. This rule is needed because, if $type(s)$ is different from $type(t)$, then there is no way —because of the dependency— that their instances can be equal.

Finally the (polymorphic) type signatures for the predicates defined in P are extracted from S_P. The extraction of the types from $\mathcal{S}_Q$ needs a small adjustment. For a predicate p defined in P, the type of its jth argument $type(p^i_j)$ is $type(p_j)\{s_1/type(s^i_1), \ldots, s_k/type(s^i_k)\}$ with $\{s_1, \ldots, s_k\} = params(type(p_j))$.

Example 8. We reconsider Example 7. P consists of the `append` clauses. $\mathcal{S}_P^{\overline{app}}$; the relevant part of the solved form is as follows:

$$\mathtt{ap_1} \supseteq \mathtt{[]} \quad \mathtt{ap_1} \supseteq \mathtt{[X|ap_1]} \quad \mathtt{ap_2} = \mathtt{ap_3} \quad \mathtt{ap_3} \supseteq \mathtt{[X|ap_3]}$$

$\mathcal{S}_Q$ consists of

$\mathtt{ap^1_1} \supseteq \mathtt{[]}$	$\mathtt{ap^1_1} \supseteq \mathtt{[X^1|ap^1_1]}$	$\mathtt{ap^1_2} = \mathtt{ap^1_3}$	$\mathtt{ap^1_3} \supseteq \mathtt{[X^1|ap^1_3]}$
$\mathtt{ap^1_1} \supseteq \mathtt{[a]}$		$\mathtt{ap^1_2} \supseteq \mathtt{[b]}$	$\mathtt{ap^1_3} = \mathtt{M}$
$\mathtt{ap^2_1} \supseteq \mathtt{[]}$	$\mathtt{ap^2_1} \supseteq \mathtt{[X^2|ap^2_1]}$	$\mathtt{ap^2_2} = \mathtt{ap^2_3}$	$\mathtt{ap^2_3} \supseteq \mathtt{[X^2|ap^2_3]}$
$\mathtt{ap^2_1} \supseteq \mathtt{[M]}$		$\mathtt{ap^2_2} \supseteq \mathtt{[M]}$	$\mathtt{ap^2_3} = \mathtt{R}$

The normal form is:

$\mathtt{ap^1_1} \supseteq \mathtt{[]}$	$\mathtt{ap^1_2} = \mathtt{ap3^1}$	$\mathtt{ap^1_3} \supseteq \mathtt{[]}$	$\mathtt{X^1} \supseteq \mathtt{a}$	$\mathtt{M} = \mathtt{ap^1_3}$
$\mathtt{ap^1_1} \supseteq \mathtt{[X^1|ap^1_1]}$		$\mathtt{ap^1_3} \supseteq \mathtt{[X^1|ap^1_3]}$	$\mathtt{X^1} \supseteq \mathtt{b}$	
$\mathtt{ap^2_1} \supseteq \mathtt{[]}$	$\mathtt{ap^2_2} = \mathtt{ap^2_3}$	$\mathtt{ap^2_3} \supseteq \mathtt{[]}$	$\mathtt{X^2} = \mathtt{ap^1_3}$	$\mathtt{R} = \mathtt{ap^2_3}$
$\mathtt{ap^2_1} \supseteq \mathtt{[X^2|ap^2_1]}$		$\mathtt{ap^2_1} \supseteq \mathtt{[X^2|ap^2_3]}$		

Rule 1 applies on $\mathtt{ap_3}$, the constraint $\mathtt{ap_3} \supseteq \mathtt{[]}$ is added to $\mathcal{S}_P$ ($\mathtt{ap^1_3} \supseteq \mathtt{[]}$ and $\mathtt{ap^2_3} \supseteq \mathtt{[]}$ are already in $\mathcal{S}_Q$) and the extracted types are:

```
ap1(X) ⟶ []; [X | ap1(X)]          append(ap1(X),ap3(X),ap3(X))
ap3(X) ⟶ []; [X | ap3(X)]
```

The signature of the first call is $\mathtt{append(type(ap^1_1),type(ap^1_2),type(ap^1_3))}$ which is an instance of the above; the instance of the type parameter `X` is given by $\mathtt{type(X^1)}$ which is `t1 ⟶ a; b`. Similarly, in the second call, the type parameter is instantiated into $\mathtt{type(X^2)} = \mathtt{type(ap^1_3)}$ which is the type `ap3(t1)`.

Example 9. This example illustrates the need for the second rule.

```
minimum(tree(X, void, Y), X).
minimum(tree(U, Left, V), W) :- minimum(Left, W).
p(S,M) :- minimum(tree(a,S,S),M).
```

[5] They are unconstrained, hence $type(r_k)$ are new type parameters in the type signature of p.

Let P consist of the first two clauses; the solved form of $\mathcal{S}_P$ is:

$$\begin{array}{llll} \texttt{min}_1 \supseteq \texttt{tree(X,min}_1\texttt{,Y)} & \texttt{min}_2 = \texttt{X} & \texttt{U = X} & \texttt{W = X} \\ \texttt{min}_1 \supseteq \texttt{void} & \texttt{Left} = \texttt{min}_1 & \texttt{V = Y} & \end{array}$$

This gives a signature with two parameters, namely `minimum(tr(X,Y),X)` with `tr(X,Y)` $\longrightarrow$ `void; tree(X,tr(X,Y),Y)`. The solved form of $\mathcal{S}_Q$ is

$$\begin{array}{lllll} \texttt{min}_1^1 \supseteq \texttt{tree(X}^1\texttt{,min}_1^1\texttt{,min}_1^1\texttt{)} & \texttt{min}_2^1 = \texttt{X}^1 & \texttt{Y}^1 = \texttt{X}^1 & \texttt{S} = \texttt{X}^1 & \\ \texttt{min}_1^1 \supseteq \texttt{void} & \texttt{M} = \texttt{X}^1 & \texttt{p}_1 = \texttt{min}_1^1 & \texttt{p}_2 = \texttt{X}^1 & \texttt{X}^1 \supseteq \texttt{a} \end{array}$$

This system implies the constraint $\texttt{Y}^1 = \texttt{min}_1^1$ while $\texttt{min}_1$ depends on `Y` in $\mathcal{S}_P$. Hence $\texttt{Y} = \texttt{min}_1$ has to be added to $\mathcal{S}_P$. For $\texttt{min}_1$, this gives the constraints $\texttt{min}_1 \supseteq \texttt{tree(X,min}_1\texttt{,min}_1\texttt{)}$ and $\texttt{min}_1 \supseteq \texttt{void}$ hence we obtain the signature `minimum(tr(X),X)` with `tr(X)` $\longrightarrow$ `void; tree(X,tr(X),tr(X))`. For $p/2$ the signature is `p(tr(t),t)` with `t` $\longrightarrow$ `a`.

6 Related Work

We can contrast this work to previous work on inferring types for logic programs in which a regular approximation of the success set (minimal Herbrand model) of a program is computed [16,25,5,11,6,22]. We derive a well-typing, which may or may not be a safe approximation of the success set. As a result our approach is not based directly on abstract interpretation, and the inference algorithm has a different structure, based on solving constraints rather than computing a fixpoint.

Our procedure resembles in some ways the set constraint approximations of logic programs developed by Heintze and Jaffar [9], as well as earlier work on deriving regular types from functional programs [18,12]. We also generate set constraints and solve them, but again, our constraints do not represent an over-approximation of the success set in contrast to the cited works. Because we aim at well-typing instead of approximating the success set, our set constraints are much simpler than those of Heintze and Jaffar. In particular there are no intersections in our set expressions, and this allows an efficient solution procedure. Marlow and Wadler [15] describe the automatic derivation of types for Erlang using a similar approach, namely the generation of set constraints capturing the well-typing requirements followed by a constraint solving procedure. Their type system is somewhat more expressive than ours, including a limited form of type complement, and the constraints generated require a more complex solution procedure. However their approach yields truly polymorphic types such that the calls are subtypes of the type signature, and thus their constraint solutions methods could be applicable in our future work in extending Section 5. Christiansen [4] also describes a method of generating type declarations that give a well-typing, using a constraint-solving approach, but his method requires some given types.

Finally we note that unlike classical type inference for ML and other typed languages, we assume no basic types. In part of [14], the authors describe (polymorphic) type reconstruction for logic programs: given a set of types and a type for each functor, they derive types for the predicates and the variables of the

program. It is noteworthy that they point out that it has been shown that the problem is undecidable unless the type of body occurrences of a recursive polymorphic predicate is *identical* to the type of the predicate (we impose this too). The main difference with our approach is that we do not provide any type definitions in advance but construct new definitions during the analysis. We share the latter property with the work in [25]; however, to our understanding, the authors do not infer parametric types - type variables are merely names for types that are defined by their own type rules - and their types are less precise than ours, since they are success set approximations.

7 Conclusion

We have presented a method for automatically deriving polymorphic well-typings for logic programs, along with its implementation and the results of some experiments. Distinguishing features of our approach are: (1) No types are assumed, the analysis constructs its own types; (2) recursive calls to a predicate are assumed to have the same type as the original call to the predicate; (3) set constraints impose only conditions for well-typing, not conditions for approximating the success set; (4) the same function symbol can be used in different type rules, i.e., a function symbol can have several type signatures. The experiments show that the inferred types are useful for termination analysis; indeed we may claim to have solved the problem of type inference for deriving norms, since we could not find any example where a user-declared type gave better termination conditions than our automatically derived types.

Future work will focus on two aspects. Firstly, we will develop the approach to polymorphism described in Section 5. Secondly we will investigate to what extent the inferred types could be used for error detection. As the procedure derives a well-typing for every program, it may seem that the possibilities are limited, but there are clear cases when the call constraints for a predicate are not consistent with any intended solution of the constraints derived from the predicate definition, and in such cases an error is indicated. For example, any call to `append` in which the first argument contains a function other than `[]` or `[.|.]` is erroneous. The exact conditions for such errors are the subject of future research.

Acknowledgements

We wish to thank Tom Schrijvers for finding errors in an earlier draft, and for useful discussions. He also verified that the generated types were accepted by the Mercury type checker.

References

1. M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis of logic programs through combination of type-based norms. Draft, 2004.

2. M. Bruynooghe, M. Codish, S. Genaim, and W. Vanhoof. Reuse of results in termination analysis of typed logic programs. In *Static Analysis, SAS 2002*, volume 2477 of *LNCS*, pages 477–492, 2002.
3. M. Bruynooghe, J. Gallagher, and W. Van Humbeeck. Inference of well-typings for logic programs with application to termination analysis. Technical Report CW 409, Dept. Comp. Sc., Katholieke Universiteit Leuven, 2005.
4. H. Christiansen. Deriving declarations from programs (extended abstract). In *CPP'97, Workshop on Constraint Programming for Reasoning about Programming, Leeds*, 1997.
5. T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Logic in Computer Science, LICS'91*, pages 300–309, 1991.
6. J. P. Gallagher and D. de Waal. Fast and precise regular approximation of logic programs. In *Logic Programming, ICLP'94*, pages 599–61, 1994.
7. J. P. Gallagher and G. Puebla. Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. In *Practical Aspects of Declarative Languages, PADL 2002*, volume 2257 of *LNCS*, pages 243–261, 2002.
8. S. Genaim, M. Codish, J. P. Gallagher, and V. Lagoon. Combining norms to prove termination. In A. Cortesi, editor, *Verification, Model Checking, and Abstract Interpretation, VMCAI 2002*, volume 2294 of *LNCS*, pages 126–138, 2002.
9. N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Principles of Programming Languages, POPL'90*, pages 197–209, 1990.
10. P. M. Hill and R. W. Topor. A semantics for typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, 1992.
11. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2-3):205–258, 1992.
12. N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Principles of Programming Languages, POPL'82*, pages 66–74, 1982.
13. V. Lagoon, F. Mesnard, and P. J. Stuckey. Termination analysis with types is more accurate. In *Logic Programming, ICLP 2003*, volume 2916 of *LNCS*, pages 254–268, 2003.
14. T. L. Lakshman and U. S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In *Logic Programming, ISLP 1991*, pages 202–217, 1991.
15. S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *ICFP*, pages 136–149, 1997.
16. P. Mishra. Towards a theory of types in Prolog. In *Logic Programming, ISLP 1984*, pages 289–298, 1984.
17. A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.
18. J. C. Reynolds. Automatic construction of data set definitions. In *Information Processing 68*, pages 456–461, 1996.
19. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.
20. C. Taboch, S. Genaim, and M. Codish. TerminWeb: Semantic based termination analyser for logic programs, 2002. http://www.cs.bgu.ac.il/ mcodish/TerminWeb.
21. R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.

22. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 22(3):179–210, 1994.
23. W. Vanhoof and M. Bruynooghe. When size does matter. In A. Pettorossi, editor, *Logic Based Program Synthesis and Transformation, LOPSTR 2001*, volume 2372 of *LNCS*, pages 129–147, 2002.
24. C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *Static Analysis, SAS 2002*, pages 102–116, 2002.
25. J. Zobel. Derivation of polymorphic types for Prolog programs. In *Logic Programming, ICLP 1987*, pages 817–838, 1987.

Memory Space Conscious Loop Iteration Duplication for Reliable Execution

G. Chen[1], M. Kandemir[1], and M. Karakoy[2]

[1] CSE Department, Penn State University, University Park, PA, USA
{guilchen, kandemir}@cse.psu.edu
[2] 180 Queen's Gate, Imperial College, London, SW7 2AZ, UK
mk22@doc.ic.ac.uk

Abstract. Soft errors, a form of transient errors that cause bit flips in memory and other hardware components, are a growing concern for embedded systems as technology scales down. While hardware-based approaches to detect/correct soft errors are important, software-based techniques can be much more flexible. One simple software-based strategy would be full duplication of computations and data, and comparing the results of the corresponding original and duplicate computations. However, while the performance overhead of this strategy can be hidden during execution if there are idle hardware resources, the memory demand increase due to data duplication can be dramatic, particularly for array-based applications that process large amounts of data.

Focusing on array-based embedded computing, this paper presents a memory space conscious loop iteration duplication approach that can reduce memory requirements of full duplication (of array data), without decreasing the level of reliability the latter provides. Our "in-place duplication" approach reuses the memory locations from the same array to store the duplicates of the elements of a given array. Consequently, the memory overhead brought by the duplicates can be reduced. Further, we extend this approach to incorporate "global duplication", which reuses memory locations from other arrays to store duplicates of the elements of a given array. This paper also discusses how our approach operates under a memory size constraint. The experimental results from our implementation show that the proposed approach is successful in reducing memory requirements of the full duplication scheme for twelve array-based applications.

1 Introduction

Soft errors, a certain type of transient errors, generally result from random electric discharges caused by background radiation, including alpha particles, cosmic rays, and nearby human sources [18,25]. The impact of a soft error on a computer system is a *bit flip* in memory components and computational logic. With the scaling of technology down into the deep-submicron range, digital circuits are even more susceptible to random failure than previous generations. If not addressed properly, soft errors can lead to dramatic problems in embedded applications from a variety of domains. For example, in safety-critical applications, unpredictable reliability can result in significant cost in terms of human and equipment loss. Similarly, in commercial consumer applications where high-volume, low-margin production is the norm, high levels of product failures

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 52–69, 2005.
© Springer-Verlag Berlin Heidelberg 2005

may necessitate the costly management of warranty support or expensive field maintenance, eventually affecting brand reputation.

Recent research has focused on the soft error problem from both architecture and software perspectives. We will discuss the related efforts in Section 6. One of the techniques that have been proposed is based on executing duplicates of instructions and comparing the results of the primary copy and duplicate to check correctness. It must be observed, however, that embedded environments typically operate under multiple constraints such as power consumption, memory size, performance and mean time to failure, and maintaining a required level of reliability against soft errors should be carefully balanced with other constraints. More specifically, one needs to consider the extra memory consumption, execution cycles, and power consumption due to duplicated instructions and data. In particular, limiting extra memory space demand of an application due to enhanced reliability is extremely important in many embedded environments. In embedded environments that execute a single application, the memory demand of the application directly determines the size of the memory to be employed, which means that an increase in memory demands can increase the overall cost of the embedded system and its area (form factor). Also, in multi-programmed embedded environments, increasing memory consumption of an application can reduce the number of applications that can execute simultaneously, thereby impacting overall performance of the system. Therefore, when increasing the number of instructions and size of data for reliability reasons, one must be careful in limiting the required extra memory space.

Motivated by this observation, this paper presents a memory space conscious loop iteration duplication scheme for reliability. The idea is to execute a copy (duplicate) of an original loop iteration (along with the original) and compare their results for correctness. In storing the results of the duplicates, we try to reuse some of the memory locations that originally store the data manipulated by the program. In other words, we recycle the memory locations as much as possible to reduce the extra memory demand due to duplicate executions. This is expected to bring two benefits. First, memory space consumption is reduced, which is very important for memory-constrained systems. Second, performance can be improved due to improved data cache behavior. Targeting array-intensive embedded applications, this paper makes the following contributions:

- We present a compiler-based approach to memory conscious loop iteration duplication. Our "in-place duplication" approach reuses memory locations from the same array to store the duplicates of its elements. Specifically, it reuses the locations of dead array elements to store the duplicates of the actively-used array elements. As a result, the memory overhead brought by duplicates is reduced.
- We discuss a "global duplication" scheme, which allows us reuse memory locations from other arrays to store the duplicates of the elements of a given array.
- We present experimental evidence demonstrating the effectiveness of the proposed approaches. Both in-place duplication and global duplication are automated within an optimizing compiler. We test our approaches using twelve array-based applications and show that in-place duplication can reduce the extra memory consumption of a duplication based scheme that does not consider memory space consumption by about 33.2%, and that the global duplication scheme brings up this figure to 42.1%.

- We demonstrate how our approach can be made to work when a limited extra memory consumption is permissible. In this scenario, our approach tries to reuse as many memory locations as possible under the specified memory constraint.

It must be emphasized that array-based codes are very important for embedded systems. This is because many embedded image and video processing programs/applications are array intensive [4], and they are usually in the form of loop nests operating on arrays of signal data.

There are several reasons why our approach is better than a hardware-based scheme, e.g., a combination of redundant instruction execution and ECC memory (i.e., memory protected by error correction code). First, if some applications (or some portions of an application) require greater reliability than others, software will be able to selectively apply duplication, instead of incurring the fixed ECC overhead on all of the memory accesses. Second, if an application needs a high level of reliability on existing hardware without ECC, a software technique would be needed. Third, our scheme can use whatever memory is available to increase reliability, i.e., we are able to decrease failure rate under a given memory space constraint.

The rest of this paper is structured as follows. In Section 1, we describe the representation used for loop iterations and array data. Section 3 discusses our assumptions, and presents our approach to in-place duplication. In the same section, we discuss our approach to duplication under a memory constraint as well. Section 4 discusses extensions to our base approach when some of our assumptions are relaxed. Section 5 gives our experimental results that show memory savings when using our approaches. In Section 6, we describe related work. Finally, in Section 7, we draw conclusions.

2 Representation for Loop Iterations, Data Space, and Array Accesses

Table 1 presents the notation used in this paper. The domain of our approach is the set of sequential array-intensive embedded programs consisting of nested loops. We assume that the loop bounds and the array indices (subscript functions) are affine functions of enclosing loop indices and loop-invariant constants. We handle other constructs such as non-affine array accesses and conditional statements conservatively.

Table 1. Notations

n	Number of enclosing loops for an array reference.
w	Number of arrays in a loop nest.
$\mathcal{I}$	Iteration space.
$\boldsymbol{I}$	$= [i_1 \quad i_2 \quad \cdots \quad i_n]^T$. An iteration point.
$\boldsymbol{I}^+$	$= \boldsymbol{I} + [0 \quad 0 \cdots 0 \quad 1]^T$.
$\preceq$	$\boldsymbol{I} \preceq \boldsymbol{J}$ means $\boldsymbol{I}$ is lexically less than or equal to $\boldsymbol{J}$.
X_k	An array.
M_k	Number of read references to X_k.
D_k	Number of dimensions of X_k.
$N_{k,i}$	Size of the ith dimension of X_k.
N_k	$= [N_{k,1} \quad N_{k,2} \ \cdots \ N_{k,D_k}]^T$. Size of X_k.
$\boldsymbol{x}$	Index of an array element.
$\mathcal{F}_{k,l}(\boldsymbol{I})$	lth reference to X_k. $\mathcal{F}_{k,l}(\boldsymbol{I}) = F_{k,l} \cdot \boldsymbol{I} + \boldsymbol{f_{k,l}}$.
R	Set of all read references in the loop body.
$\mathcal{G}_k(R)$	Right hand side of the kth statement.
$\boldsymbol{d_{k,l}}$	Dependence distance from $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ to $X_k(\mathcal{F}_{k,l}(\boldsymbol{I}))$; that is, $X_k(\mathcal{F}_{k,0}(\boldsymbol{I})) = X_k(\mathcal{F}_{k,l}(\boldsymbol{I} + \boldsymbol{d_{k,l}}))$
$\boldsymbol{d_{k,l_{max}}}$	$\max_{1 \le l \le M_k} \boldsymbol{d_{k,l}}$; that is, the maximum reuse distance (in terms of lexicographical order) from $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ to $X_k(\mathcal{F}_{k,l}(\boldsymbol{I}))$.
$\boldsymbol{L_k}$	Iteration offset for duplicates. The duplicate of $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ is stored in $X_k(\mathcal{F}_{k,0}(\boldsymbol{I} + \boldsymbol{L_k}))$.
$\boldsymbol{P_k}$	$= [p_{k,1} \quad p_{k,2} \ \cdots \ p_{k,D_k}]^T$. Space offset for duplicates. The duplicate of $X_k(\boldsymbol{x})$ is stored in $X_k(\boldsymbol{x} + \boldsymbol{P_k})$.
$\lvert p_{k,i} \rvert$	Absolute value of $p_{k,i}$.
INC_k	Array size expansion of X_k.

```
for i = 0,N-1
  for j = 1,N-1
    A(i,j) = 2*A(i,j-1) + 1;
```

Fig. 1. An example nested loop

In a given loop nest with n loops, iterators surrounding any statement can be represented as an n-entry vector $\boldsymbol{I} = [i_1 \quad i_2 \quad \cdots \quad i_n]^T$. The iteration space $\mathcal{I}$ consists of all the iterations of a loop nest. We use $\boldsymbol{I}^+$ as a shorthand for $\boldsymbol{I} + [0 \quad 0 \ \cdots \ 0 \quad 1]^T$. The index domain of an m-dimensional array X_k is a rectilinear polyhedron, in which each element can be represented as an m-entry vector $\boldsymbol{x_k} = [a_1 \quad a_2 \quad \cdots \quad a_m]^T$. We use $\mathcal{F}_{k,l}(\boldsymbol{I})$ to represent the access function of the lth reference to array X_k. $\mathcal{F}_{k,l}(\boldsymbol{I})$ can also be defined in a matrix/vector form as: $\mathcal{F}_{k,l}(\boldsymbol{I}) = F_{k,l} \cdot \boldsymbol{I} + \boldsymbol{f_{k,l}}$, where $F_{k,l}$ is an $m \times n$ matrix and $\boldsymbol{f_{k,l}}$ is an m-entry vector. As an example, for the two references shown in Fig. 1, we have:

$$\mathcal{F}_{1,0}(\boldsymbol{I}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ and } \mathcal{F}_{1,1}(\boldsymbol{I}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}.$$

A *data reuse* is said to exist from an array reference $\mathcal{F}_{k,l_1}$ to an array reference $\mathcal{F}_{k,l_2}$ if: $\exists \boldsymbol{I_1} \in \mathcal{I}, \boldsymbol{I_2} \in \mathcal{I} : \boldsymbol{I_1} \preceq \boldsymbol{I_2}$ and $\mathcal{F}_{k,l_1}(\boldsymbol{I_1}) = \mathcal{F}_{k,l_2}(\boldsymbol{I_2})$. In this case, $\boldsymbol{I_2} - \boldsymbol{I_1}$ is defined as the *reuse distance* between $\mathcal{F}_{k,l_1}$ and $\mathcal{F}_{k,l_2}$. For example, in Fig. 1, data reuse exists from $\mathcal{F}_{k,0}$ to $\mathcal{F}_{k,1}$ since $\mathcal{F}_{k,0}(\boldsymbol{I}) = \mathcal{F}_{k,1}(\boldsymbol{I}^+)$, and the reuse distance between them is $[0 \quad 1]^T$.

3 Array Duplication

A simple approach to enhance reliability is to create a duplicated copy for each array, duplicate the execution of each iteration, and compare the result of the primary with that of the duplicate. We refer to this approach as *full duplication* in this paper. An important problem with this approach is that it doubles the memory space consumption (as each array is duplicated). Our objective is to improve the reliability of the computation in the loop, and keep the incurred memory cost at minimum. We achieve this by not duplicating the array fully, but *reusing* some memory locations (that are used to store other elements) for duplicates.

3.1 Assumptions

Our algorithm works on a per-loop basis. We assume that all the loops are normalized, i.e., the loop index variable of each loop nest increases by 1 at each step. Loop normalization [1] is a standard code modification technique that can be used to ensure this. In this section, we consider "in-place duplication", which means reusing array elements (i.e., their memory locations) for storing the duplicates of the elements of the same array. That is, for an array element, its duplicate can be stored only within the same array. In Section 4.2 we present the algorithm that allows "global duplication", i.e., reusing array locations for storing the duplicates of the elements from other arrays. Our algorithm operates on one array at a time. For an array X_k to be considered as a candidate by our algorithm, the following assumptions must be satisfied:

- **Assumption 1:** For every pair of array references to X_k, the reuse distance between them is a constant vector. Note that if two array references do not have any data reuse between them, they are also assumed to have a constant reuse distance vector. Most existing compiler optimizations for array-based codes operate under this assumption.
- **Assumption 2:** If an array element of X_k is written in the loop nest, all the reads to this array element retrieve the value stored by some write reference in the loop nest (that is, none of the reads to this element retrieves a value stored before the loop nest).
- **Assumption 3:** There is only one write reference to X_k in the loop body.

Whether an array satisfies Assumption 1 and Assumption 2 can be checked using data reuse analysis [23] and value dependence test [11]. Checking Assumption 3 is straightforward. In Section 4, we discuss the cases where we relax these assumptions. In in-place duplication, if an array does not satisfy all of the above assumptions, we fall back to the full duplication strategy for that array. For now, let us assume that all the arrays in the loop satisfy these assumptions. Based on the assumptions above, a loop body with w arrays can be represented as:

$$\begin{aligned} X_1(\mathcal{F}_{1,0}(\boldsymbol{I})) &= \mathcal{G}_1(R); \\ X_2(\mathcal{F}_{2,0}(\boldsymbol{I})) &= \mathcal{G}_2(R); \\ &\vdots \\ X_w(\mathcal{F}_{w,0}(\boldsymbol{I})) &= \mathcal{G}_w(R). \end{aligned}$$

R is the set of all read array references in the loop body, and $\mathcal{G}_i$ ($1 \leq i \leq w$) represents a function of these read references. In mathematical terms:

$$\begin{aligned} R = \{ \; & X_1(\mathcal{F}_{1,1}(\boldsymbol{I})), X_1(\mathcal{F}_{1,2}(\boldsymbol{I})), \cdots, X_1(\mathcal{F}_{1,M_1}(\boldsymbol{I})), \\ & X_2(\mathcal{F}_{2,1}(\boldsymbol{I})), X_2(\mathcal{F}_{2,2}(\boldsymbol{I})), \cdots, X_2(\mathcal{F}_{2,M_2}(\boldsymbol{I})), \\ & \vdots \\ & X_w(\mathcal{F}_{w,1}(\boldsymbol{I})), X_w(\mathcal{F}_{w,2}(\boldsymbol{I})), \cdots, X_w(\mathcal{F}_{w,M_w}(\boldsymbol{I})) \; \}. \end{aligned}$$

$\mathcal{F}_{k,0}$ is the write reference to array X_k, and M_k is the number of read references to X_k.

Based on these assumptions, we can determine that there is a reuse from $\mathcal{F}_{k,0}$ to each read reference $\mathcal{F}_{k,l}$, and the corresponding reuse distance is a constant vector, which we denote using $\boldsymbol{d_{k,l}}$. This means that the array element $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$, which is written at iteration $\boldsymbol{I}$, will be used at iterations $\boldsymbol{I}+\boldsymbol{d_{k,1}}$, $\boldsymbol{I}+\boldsymbol{d_{k,2}}$, ..., $\boldsymbol{I}+\boldsymbol{d_{k,M_k}}$. This can be also expressed as:

$$\mathcal{F}_{k,l}(\boldsymbol{I}+\boldsymbol{d_{k,l}}) = \mathcal{F}_{k,0}(\boldsymbol{I}).$$

Let us assume that $\mathcal{F}_{k,l_{max}}$ is the one with the maximum reuse distance (in terms of lexicographical order) from $\mathcal{F}_{k,0}$; that is, $\boldsymbol{d_{k,l_{max}}} = \max_{1 \leq l \leq M_k}(\boldsymbol{d_{k,l}})$. Therefore, $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ written at iteration $\boldsymbol{I}$ is *last-used* at iteration $\boldsymbol{I}+\boldsymbol{d_{k,l_{max}}}$ by array reference $\mathcal{F}_{k,l_{max}}$.

3.2 In-place Duplication

Approach and Algorithm. For the execution of a statement to be reliable, we need to duplicate its input data, duplicate its execution, and compare the results of the original and duplicated executions. For example, the reliable version of a statement "`A(i)=`" "`A(i-1)+1;`" would be:

```
A(i) = A(i-1) + 1;
A'(i) = A'(i-1) + 1;
if A(i) != A'(i)
  error();
```

We assume that, `A'` is a duplicate for array `A` in the above statement. In this section, we discuss how we reduce the memory space overhead brought by duplicates without compromising reliability.

A memory location can be in two different states: *active* or *inactive*. At a given time, a memory location is "active" if the value stored in it will be used in the future. On the other hand, a memory location is "inactive" if there is not any future read operation on it, or its value is updated before any read operation on it takes place. As an example, Fig. 2 gives the states of three variables, a, b and c, at different points of time during execution. At any given time, we need to provide a duplicate for an active array element, so that any soft error that occurs in its location can be detected by comparing this array element and its duplicate. It is to be noted that, we can modify the value in an inactive location without affecting the correctness of the program. Therefore, the inactive memory locations are good candidates for storing the duplicates of the active memory locations. For example, in Fig. 2, we can use variable c to store the duplicate of a, because variable c is inactive from t_1 to t_2 and from t_4 to t_6, during which a is active, and needs to be duplicated if it is to be protected against soft errors. In this section, we focus on "in-place duplication", which means reusing inactive array elements (i.e., their locations) for storing the duplicates of the elements of the same array.

Let us consider an array X_k. Fig. 3 illustrates a scenario for selecting the location to store the duplicate for an element updated at loop iteration $\boldsymbol{I}$. At iteration $\boldsymbol{I}$, $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ is updated, and the same array element is last-used at iteration $\boldsymbol{J}$. Consequently, the array element $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ is active between iterations $\boldsymbol{I}$ and $\boldsymbol{J}$, and we need to keep a duplicate for it during this period. To save memory space, we want to find an element in X_k, which is inactive during this period. Recall that Assumption 2 presented in Section 3.1 says that all the read references to an array element are executed after the corresponding write reference (if such a write reference exists). Therefore, if an element is written in some loop iteration, it is inactive before that iteration. Consequently, if an array element is written after iteration $\boldsymbol{J}$, the last iteration at which

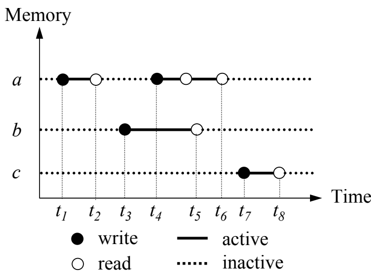


Fig. 2. States of memory locations during execution

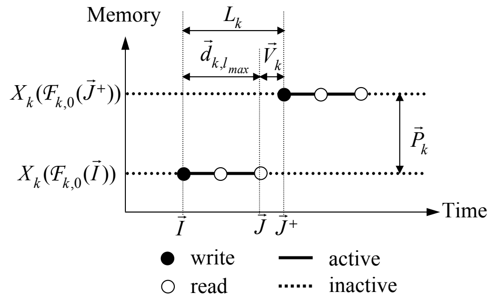


Fig. 3. Determining a memory location to store the duplicate for element $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$

$X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ is used, this element can be used to store the duplicate of $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$, since it is inactive between $\boldsymbol{I}$ and $\boldsymbol{J}$. Our approach uses the array element written at iteration $\boldsymbol{J} + \boldsymbol{V_k}$, which is $\mathcal{F}_{k,0}(\boldsymbol{J} + \boldsymbol{V_k})$, to store the duplicate of the array element written at iteration $\boldsymbol{I}$. Here, $\boldsymbol{V_k}$ is a constant vector and $[0 \quad 0 \cdots 0 \quad 0]^T \prec \boldsymbol{V_k}$ so that iteration $\boldsymbol{J} + \boldsymbol{V_k}$ is executed after iteration $\boldsymbol{J}$. A possible choice for $\boldsymbol{V_k}$ will be discussed shortly.

Based on the discussion in Section 3.1, we know that $\boldsymbol{J} = \boldsymbol{I} + \boldsymbol{d_{k,l_{max}}}$. We use $\boldsymbol{L_k}$ to represent the distance between $\boldsymbol{I}$ and $\boldsymbol{J} + \boldsymbol{V_k}$. Hence, we have:

$$\boldsymbol{L_k} = \boldsymbol{J} + \boldsymbol{V_k} - \boldsymbol{I} = \boldsymbol{d_{k,l_{max}}} + \boldsymbol{V_k}.$$

Thus, the duplicate of $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ is stored in $X_k(\mathcal{F}_{k,0}(\boldsymbol{I} + \boldsymbol{L_k}))$. Consequently, the memory space distance between these two elements, denoted as $\boldsymbol{P_k}$, can be calculated as:

$$\begin{aligned}\boldsymbol{P_k} &= \mathcal{F}_{k,0}(\boldsymbol{I} + \boldsymbol{L_k}) - \mathcal{F}_{k,0}(\boldsymbol{I}) \\ &= (F_{k,0} \cdot (\boldsymbol{I} + \boldsymbol{L_k}) + f_{k,0}) - (F_{k,0} \cdot \boldsymbol{I} + f_{k,0}) \\ &= F_{k,0} \cdot (\boldsymbol{I} + \boldsymbol{L_k}) - F_{k,0} \cdot \boldsymbol{I} \\ &= F_{k,0} \cdot \boldsymbol{L_k}.\end{aligned}$$

Note that $\boldsymbol{P_k}$ is a constant vector since both $F_{k,0}$ and $\boldsymbol{L_k}$ are constant vectors. Consequently, for an arbitrary array element $X_k(\boldsymbol{x})$, its duplicate can reside in $X_k(\boldsymbol{x} + \boldsymbol{P_k})$, and this process can be carried out for every array used in the loop nest.

It should be observed that if $\boldsymbol{x}$ is near the array boundary, $\boldsymbol{x} + \boldsymbol{P_k}$ may exceed the original array boundary. In this case, we need to expand array X_k so that $\boldsymbol{x} + \boldsymbol{P_k}$ remains within the boundary. Assuming that $\boldsymbol{P_k} = [p_{k,1} \quad p_{k,2} \quad \cdots \quad p_{k,D_k}]^T$ and that the original size of the ith dimension of X_k is $N_{k,i}$, the ith dimension of X_k needs to be expanded by $|p_{k,i}|$, units (i.e., array elements) to $N_{k,i} + |p_{k,i}|$ (we use $|p_{k,i}|$ to denote the absolute value of $p_{k,i}$). Therefore, the total memory expansion for array X_k, denoted as INC_k, can be calculated as:

$$INC_k = \prod_{i=1}^{D_k}(N_{k,i} + |p_{k,i}|) - \prod_{i=1}^{D_k} N_{k,i} \tag{1}$$

Note that we expect INC_k to be much smaller than $\prod_{i=1}^{D_k} N_{k,i}$, the total size of the array. Let us now look at the problem of how to select a suitable V_k. Since our objective is to minimize the memory consumption due to duplication, we want to select a $\boldsymbol{V_k}$ so that INC_k can be minimized. Although an optimum $\boldsymbol{V_k}$ can be calculated by exhaustive enumeration or other sophisticated methods, we use a simple heuristic here that sets $\boldsymbol{V_k}$ to $[0\ 0\ \cdots\ 0\ 1]^T$. The rationale behind this choice is that by minimizing V_k, we can minimize $\boldsymbol{P_k}$, and, thus, we can minimize INC_k. More specifically, in this case, we obtain:

$$\boldsymbol{L_k} = \boldsymbol{d_{k,l_{max}}} + [0 \quad 0 \cdots 0 \quad 1]^T = \boldsymbol{d_{k,l_{max}}}^+.$$

A potential problem is that $p_{k,i}$ could be negative for some i, which means that $x_i + p_{k,i}$ can be a negative number, where x_i is the array index of the original array reference for the ith dimension. Such a case can arise if the array is accessed from upper to lower index along the ith dimension. If this is the case, we use "$(x_i + p_{k,i} + N_{k,i}) \bmod N_{k,i}$" as the array index for this dimension. That is, we use the additional (upper) elements for placeholders of the duplicates of the lower elements.

Our algorithm for in-place duplication is given in Fig. 4. Assume that there are K arrays in the loop body, the average number of references to each array is M, the average

Algorithm I:

foreach array X_k do
  check the following three assumptions:
    there is only one write reference to X_k;
    each read reference has a data reuse from the write reference;
    the reuse distances are constant vectors;
  if all assumptions are satisfied
    $\boldsymbol{d_{k,l_{max}}} = \max_{1 \le l \le M_k} \boldsymbol{d_{k,l}}$;
    $\boldsymbol{L_k} = \boldsymbol{d_{k,l_{max}}} + [0 \;\; 0 \cdots 0 \;\; 1]^T$
    $\boldsymbol{P_k} = F_{k,0} \cdot \boldsymbol{L_k}$;
    foreach dimension i of array X_k do
      $N_{k,i}$ += $|p_{k,i}|$;
    endfor
    foreach reference $X_k(\mathcal{F}_{k,l}(\boldsymbol{I}))$ do
      its duplicate is stored in
        $X_k(\mathcal{F}_{k,l}(\boldsymbol{I}) + \boldsymbol{P_k})$;
    endfor
  else use full duplication for X_k;
  endif
endfor

Fig. 4. Algorithm I: The algorithm for in-place duplication

Algorithm II:

foreach array X_k do
  calculate INC_k;
endfor
sort the arrays as $X_{k_1}, X_{k_2}, \ldots, X_{k_w}$,
so that $INC_{k_1} \le INC_{k_2} \le \cdots \le INC_{k_w}$;
h = 1;
Mem = 0;
while $h \le w$ do
  if $(Mem + INC_{k_h}) \le U$ do
    Mem += INC_{k_h};
    h++;
  else goto LoopExit;
  endif
endwhile
LoopExit:
h = h - 1;
for i = 1, h do
  duplicate X_{k_i};
endfor

Fig. 5. Algorithm II: The algorithm for selecting the arrays to duplicate under memory constraint (U)

```
int A(N);
for i=0,N-2
  A(i+1)=A(i)+a;
```

(a) Original program

```
int A(N),A'(N);
for i=0,N-2 {
  A(i+1)=A(i)+a;
  A'(i+1)=A'(i)+a;
  if A(i+1)!=A'(i+1)
    error();
}
```

(b) Full duplication

```
int A(N+2);
for i=0,N-2 {
  A(i+1)=A(i)+a;
  A(i+3)=A(i+2)+a;
  if A(i+1)!=A(i+3)
    error();
}
```

(c) In-place duplication

Fig. 6. Example application of in-place duplication

number of dimensions of each array is D, and the number of enclosing loops is n. Apart from checking our three assumptions, for each array X_k, the time to calculate $\boldsymbol{d_{k,l}}$ and $\boldsymbol{d_{k,l_{max}}}$ is $O(MD)$. It takes $O(nD)$ time to calculate $\boldsymbol{P_k}$. Therefore, the complexity of our algorithm, without taking into account the complexity of checking assumptions, is $O((M+n)DK)$. The time for checking our three assumptions is determined by the algorithm used for value dependence testing.

Example. We now discuss an example to illustrate our in-place duplication algorithm. Fig. 6 gives the example for our algorithm written in a pseudo-language syntax. In this figure, a and N are constants. In Fig. 6, we have:

$$F_{1,0} = [1]; \;\; \mathcal{F}_{1,0}(\boldsymbol{I}) = i+1; \;\; F_{1,1} = [1]; \;\; \mathcal{F}_{1,1}(\boldsymbol{I}) = i.$$

It is easy to determine that array A satisfies the three assumptions in Algorithm I, and we have $\boldsymbol{d_{1,1}} = [1]$. Therefore, we can obtain $\boldsymbol{d_{1,l_{max}}}$ as:

$$\boldsymbol{d_{1,l_{max}}} = \boldsymbol{d_{1,1}} = [1].$$

Based on this, we can calculate $\boldsymbol{L_1}$ and $\boldsymbol{P_1}$ as follows:

$$\boldsymbol{L_1} = \boldsymbol{d_{1,l_{max}}} + [1] = [2] \quad \text{and} \quad \boldsymbol{P_1} = F_{1,0} \cdot \boldsymbol{L_1} = [2].$$

Thus, we determine that we need to expand the original memory space allocated for array `A` by 2 elements. As a result, the duplicate of `A(i+1)` is in `A(i+1+2)`, which is `A(i+3)`, and the duplicate of `A(i)` is in `A(i+2)`. Using full duplication shown in Fig. 6(b), the total size of memory is increased by 100% over the original case with no duplication. In comparison, using our in-place duplication version in Fig. 6(c), the percentage memory increase over the original case is `2/N`, which is less than 2% when $\texttt{N} > 100$.

3.3 Duplication Under Memory Constraint

Approach and Algorithm. There exist cases where one may want to limit the memory consumption brought by duplication to a certain value. In this part, we discuss how our approach can be made to work under such a memory size constraint.

We assume that all the array elements are of equal importance (as far as improving reliability against soft errors is concerned), and our objective is to have duplicates for as many array elements as possible. Let us assume that we cannot reserve more than U units (array elements) of memory space to store duplicates. From Algorithm I and Equation (1), we can calculate the memory expansion for arrays that can make use of in-place duplication. On the other hand, for an array that needs to be fully duplicated, the incurred extra memory expansion is equal to its original size. In either case, we are able to determine INC_k for each array X_k. Next, we sort our arrays according to non-decreasing INC_k values, that is:

$$X_{k_1},\ X_{k_2},\ \ldots, X_{k_w},\ \text{ where } INC_{k_1} \le INC_{k_2} \le \cdots \le INCk_w.$$

After that, we determine a maximum h such that $h \le w$ and $\sum_{i=1}^{h} INC_{k_i} \le U$. That is, we choose the candidate arrays for duplication in the increasing order of INC_k, until all the arrays are duplicated or duplicating more arrays would exceed the allowable memory size constraint. Here, h is the number of arrays that we choose during this process. Fig. 5 (on page 59) gives the algorithm (named Algorithm II) that selects the arrays to duplicate. After the selection is performed, we use the algorithm in Fig. 4 to duplicate the selected arrays.

Example. An example of duplication under memory constraint is shown in Fig. 7. By checking array `A` and array `B`, in Fig. 7(a) against our assumptions, we can determine that `A` can use in-place duplication and `B` needs to be fully duplicated. If there is no memory constraint, the original program could be transformed to the one given in

```
int A(100),B(100);
for i=0,98
  A(i+1)=A(i)+a;
  B(i+1)=B(i+1)+B(i);
```

(a) Original program

```
int A(102),B(100),B'(100);
for i=0,98
  A(i+1)=A(i)+a;
  A(i+3)=A(i+2)+a;
  if A(i+1)!=A(i+3)
    error();
  B(i+1)=B(i+1)+B(i);
  B'(i+1)=B'(i+1)+B'(i);
  if B(i+1)!=B'(i+1);
    error();
```

(b) Without memory constraint

```
int A(102),B(100);
for i=0,98
  A(i+1)=A(i)+a;
  A(i+3)=A(i+2)+a;
  if A(i+1)!=A(i+3)
    error();
  B(i+1)=B(i+1)+B(i);
```

(c) With an allowable increase of 10 array locations

Fig. 7. Example for duplication under memory constraint

Fig. 7(b). Now let us assume that we impose a memory constraint such that we cannot use more than 10 extra array locations for storing the duplicates. We use X_1 to represent array A and X_2 to represent array B. To determine the memory expansion due to array A, we proceed as follows:

$$F_{1,0} = [1];\ \mathcal{F}_{1,0}(\boldsymbol{I}) = i+1;\ F_{1,1} = [1];\ \mathcal{F}_{1,1}(\boldsymbol{I}) = i;\ \boldsymbol{d_{1,1}} = [1];\ \boldsymbol{d_{1,l_{max}}} = \boldsymbol{d_{1,1}} = [1];$$
$$\boldsymbol{L_1} = \boldsymbol{d_{1,l_{max}}} + [1] = [2];\ \ \boldsymbol{P_1} = F_{1,0} \cdot \boldsymbol{L_1} = [2];\ \ INC_1 = (100+2) - 100 = 2.$$

On the other hand, B needs to be fully duplicated. Thus, we have $INC_2 = 100$. Since $INC_1 < INC_2$, we first consider duplicating A, which is possible since $INC_1 < 10$. However, we cannot add B to the list of arrays to be duplicated since $INC_1 + INC_2 > 10$. To sum up, A is duplicated, whereas B is not duplicated. Fig. 7(c) gives the transformed code.

4 Extensions

Recall that, in Section 3.1, we listed three assumptions so that our in-place duplication could be used. In this section, we discuss the needed extensions to our base approach if some of these assumptions are to be relaxed. Note that Assumption 1 cannot be relaxed, since our approach would not work on an array that does not satisfy this assumption (i.e., if this assumption fails, we cannot put an upper bound on the extra memory space required). On the other hand, our approach can be extended to work on arrays that do not satisfy Assumption 2 or Assumption 3 (instead of just using full duplication for them).

4.1 Relaxing Assumption 3

Assumption 3 presented in Section 3.1 requires that there is only one write reference to the array being considered. Let us now consider the case where there are two write references, $\mathcal{F}_{k,0}$ and $\mathcal{F}_{k,1}$, for the array X_k being considered, and X_k satisfies Assumption 1 and Assumption 2. In this case, there are two possible scenarios for these two write references: either there is a data reuse between them, or there is no data reuse between them.

If there is a data reuse between these two write references, our algorithm can deal with this case with little modification. Without loss of generality, we assume that there is a data reuse from $\mathcal{F}_{k,0}$ to $\mathcal{F}_{k,1}$, and the reuse distance vector is $\boldsymbol{d_{k,1}}$. That is, $\mathcal{F}_{k,1}(\boldsymbol{I} + \boldsymbol{d_{k,1}}) = \mathcal{F}_{k,0}(\boldsymbol{I})$ $(\boldsymbol{d_{k,1}} \succeq [0\ \ 0\ \ \cdots\ \ 0]^T)$. This scenario is illustrated in Fig. 8. Comparing Fig. 2 and Fig. 8, we see that one can use the same strategy in determining the location to store the duplicate for $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$. In fact, we can treat $\mathcal{F}_{k,1}$ the same way as we treat read references. This is because we are certain that $X_k(\mathcal{F}_{k,0}(\boldsymbol{J}^+))$ is not touched in the original loop until iteration $\boldsymbol{J}^+$ executes.

On the other hand, if there is no data reuse between these two write references, one can treat them as two different arrays. In this case, the references to X_k can be divided into two groups, based on to which write reference they have data reuse:

$$\mathcal{F}^1_{k,0},\ \mathcal{F}^1_{k,1},\ \ldots,\ \mathcal{F}^1_{k,M^1_k};$$
$$\mathcal{F}^2_{k,0},\ \mathcal{F}^2_{k,1},\ \ldots,\ \mathcal{F}^2_{k,M^2_k}.$$

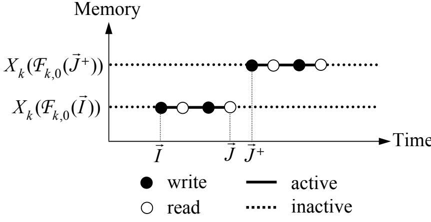


Fig. 8. Determining the memory location to store the duplicate for $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ when there are two write references to X_k

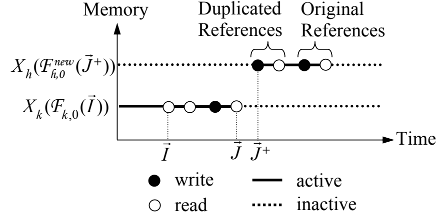


Fig. 9. Determining the memory location to store the duplicate for $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ in another array X_h

```
int A(N),B(N);
for i=0,N-2
  A(i+1)=A(i)+a;
  B(i+1)=A(i+1)+B(i);
  A(i)=B(i+1)/2;
```

(a) Original program

```
int A(N),A'(N),B(N),B'(N);
for i=0,N-2 {
  A(i+1)=A(i)+a;
  A'(i+1)=A'(i)+a;
  if A(i+1)!=A'(i+1)
    error();
  B(i+1)=A(i+1)+B(i);
  B'(i+1)=A'(i+1)+B'(i);
  if B(i+1)!=B'(i+1)
    error();
  A(i)=B(i+1)/2;
  A'(i)=B'(i+1)/2;
  if A(i)!=A'(i)
    error();
}
```

(b) Full duplication

```
int A(N+2),B(N+2);
for i=0,N-2 {
  A(i+1)=A(i)+a;
  A(i+3)=A(i+2)+a;
  if A(i+1)!=A(i+3)
    error();
  B(i+1)=A(i+1)+B(i);
  B(i+3)=A(i+3)+B(i+2);
  if B(i+1)!=B(i+3)
    error();
  A(i)=B(i+1)/2;
  A(i+2)=B(i+3)/2;
  if A(i)!=A(i+2)
    error();
```

(c) In-place duplication

Fig. 10. Example for multiple write references to the same array

Notice that there is a data reuse from $\mathcal{F}^i_{k,0}$ to $\mathcal{F}^i_{k,l}$ for $i = 1, 2$. The array elements accessed by these two groups do not overlap (since, otherwise, $\mathcal{F}^1_{k,0}$ and $\mathcal{F}^2_{k,0}$ would have data reuse); therefore, choosing an array element within one group as the location of a duplicate does not affect any access in the other group. This essentially means that we can treat the two groups as two different arrays, and select the locations for duplicates independently. The only modification to Algorithm I would be combining the array expansion results from these two groups together. For example, if our approach requires expanding the ith dimension of X_k by $|p^1_{k,i}|$ for the first group, and by $|p^2_{k,i}|$ for the second group, the final result is that the ith dimension is expanded by $max(|p^1_{k,i}|, |p^2_{k,i}|)$.

If there are more than two write references to array X_k in the loop, we can deal with them in a similar fashion. Specifically, we first divide the references into groups such that the references (in the same group) have data reuses between them, and the references in different groups are independent from each other. Then, we process each group separately as if it is a different array. Specifically, for each group X^i_k, we determine the write reference that have data reuse to all other references in its group and the reuse distances are non-negative. We represent such write reference as $\mathcal{F}^i_{k,0}$. This can also be expressed as $\forall 0 \leq l \leq M^i_k : \mathcal{F}^i_{k,l}(\boldsymbol{I} + \boldsymbol{d^i_{k,l}}) = \mathcal{F}^i_{k,0}(\boldsymbol{I})$ and $\boldsymbol{d_{k,l}} \succeq [0\ 0\ \cdots\ 0]^T$. After this, we can process this group using Algorithm I. Fig. 10 gives an example of

how in-place duplication works when there are two write references in the same loop to the same array. There are two different arrays accessed in the code. Fig. 10(b) gives the full duplication version. Assume that X_1 represents `A` and X_2 represents `B`. Array `B` satisfies all the assumptions, and we can apply in-place duplication to it using Algorithm I. On the other hand, Array `A` satisfies Assumption 1 and Assumption 2, but does not satisfy Assumption 3 since there are two write references to it (`A(i+1)` and `A(i)`). Consequently, we need to use the strategy discussed above for in-place duplication for array `A`.

For the two write references to array `A`, namely, `A(i+1)` and `A(i)`, we can determine that `A(i)` has data reuse with `A(i+1)` based on data reuse analysis. Therefore, we represent `A(i+1)` as $X_1(\mathcal{F}_{1,0}(\boldsymbol{I}))$. Now, we can apply Algorithm I to `A`:

$$F_{1,0} = [1];\ \mathcal{F}_{1,0}(\boldsymbol{I}) = i+1;\ F_{1,1} = [1];\ \mathcal{F}_{1,1}(\boldsymbol{I}) = i;\ F_{1,2} = [1];\ \mathcal{F}_{1,2}(\boldsymbol{I}) = i+1;$$
$$F_{1,3} = [1];\ \mathcal{F}_{1,3}(\boldsymbol{I}) = i; \boldsymbol{d_{1,1}} = [1];\ \boldsymbol{d_{1,2}} = [0];\ \boldsymbol{d_{1,3}} = [1];$$
$$\boldsymbol{d_{1,l_{max}}} = max(\boldsymbol{d_{1,1}}, \boldsymbol{d_{1,2}}, \boldsymbol{d_{1,3}}) = \boldsymbol{d_{1,1}} = [1].$$

Based on this, we can calculate $\boldsymbol{L_1}$ and $\boldsymbol{P_1}$ as follows:

$$\boldsymbol{L_1} = \boldsymbol{d_{1,l_{max}}} + [1] = [2] \quad \text{and} \quad \boldsymbol{P_1} = F_{1,0} \cdot \boldsymbol{L_1} = [2].$$

Therefore, we find that the duplicate of `A(i+1)` is stored in `A(i+3)`, and the duplicate of `A(i)` is stored in `A(i+2)`. Fig. 10(c) gives the transformed code when both `A` and `B` are duplicated using in-place duplication.

4.2 Relaxing Assumption 2: Global Duplication

If an array X_k does not satisfy Assumption 2, this means that there exist some array elements that are used before they are written in the loop. Such locations need to be considered active from the beginning of the loop, and we cannot use them as duplicates for other array elements. Therefore, we are not able to use in-place duplication for such an array. However, as long as X_k satisfies Assumption 1, it is still possible to avoid full duplication using a different approach, which we discuss in this subsection. This approach reuses the locations in some other array (X_h) to store the duplicates for X_k, and is referred to as "global duplication".

For an array X_h to be used to store the duplicates for X_k, it needs to satisfy the following two conditions:

1. X_h should have the same number of dimensions as X_k.
2. X_h should satisfy all the three assumptions listed in Section 3.1.

Note that such an array X_h itself can benefit from in-place duplication, and in our approach, we always apply in-place duplication first. Therefore, when we try to use the locations in X_h to store duplicates of the elements of X_k, we need to take X_h's in-place duplication into account as well. Fig. 9 illustrates an example scenario. After X_h's in-place duplication, the references to X_h are doubled due to references to duplicates. We use $\mathcal{F}^{new}_{h,l}$ to denote both the original references and the references created by duplication. We have:

$$\begin{aligned} \mathcal{F}_{h,l}(\boldsymbol{I}) + \boldsymbol{P_h} &= \mathcal{F}^{new}_{h,l}(\boldsymbol{I}), && \text{for duplicated references;} \\ \mathcal{F}_{h,l}(\boldsymbol{I}) &= \mathcal{F}^{new}_{h,l+M_h}(\boldsymbol{I}), && \text{for original references;} \\ M^{new}_h &= 2M_h. \end{aligned}$$

```
int A(N),B(N);
for i=0,N-2
  A(i+1)=A(i)+a;
  B(i+1)=B(i+1)+B(i);
```

(a) Original program

```
int A(N+2),B(N),B'(N);
for i=0,N-2 {
  A(i+1)=A(i)+a;
  A(i+3)=A(i+2)+a;
  if A(i+1)!=A(i+3)
    error();
  B(i+1)=B(i+1)+B(i);
  B'(i+1)=B'(i+1)+B'(i);
  if B(i+1)!=B'(i+1)
    error();
}
```

(b) In-place duplication

```
int A(N+4),B(N);
for i=0,N-2 {
  A(i+1)=A(i)+a;
  A(i+3)=A(i+2)+a;
  if A(i+1)!=A(i+3)
    error();
  B(i+1)=B(i+1)+B(i);
  A(i+5)=A(i+5)+A(i+4);
  if B(i+1)!=A(i+5)
    error();
}
```

(c) Global duplication

Fig. 11. Example application of global duplication

For simplicity, we assume that all references to X_k have data reuses with each other. (if this assumption is not satisfied, we use the strategy discussed in Section 4.1 by dividing references into groups). In this case, we can find a reference, denoted as $\mathcal{F}_{k,0}$, from which all other X_k references have data reuses. We follow the approach described in Algorithm I to calculate $\boldsymbol{d_{k,l_{max}}}$ and $\boldsymbol{L_k}$. We know at this point that the array element $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ will not be used from iteration $\boldsymbol{I}+\boldsymbol{L_k}$ onwards. Therefore, we can use the X_h array element $X_h(\mathcal{F}_{h,0}^{new}(\boldsymbol{I}+\boldsymbol{L_k}))$, which is written at iteration $\boldsymbol{I}+\boldsymbol{L_k}$ for the first time in the loop, to store the duplicate for $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$. To calculate the location of the duplicate for $X_k(\mathcal{F}_{k,i}(\boldsymbol{I}))$, we first represent it as $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}-\boldsymbol{d_{k,i}}))$. Therefore, the duplicate of $X_k(\mathcal{F}_{k,i}(\boldsymbol{I}))$ is stored in $X_h(\mathcal{F}_{h,0}^{new}(\boldsymbol{I}-\boldsymbol{d_{k,i}}+\boldsymbol{L_k}))$.

In order to determine how much X_h needs to be expanded, we calculate $\boldsymbol{P_k}$, i.e., the difference between $\mathcal{F}_{h,0}^{new}(\boldsymbol{I}+\boldsymbol{L_k})$ and $\mathcal{F}_{h,0}^{new}(\boldsymbol{I})$:

$$P_k = \mathcal{F}_{h,0}^{new}(\boldsymbol{I}+\boldsymbol{L_k}) - \mathcal{F}_{h,0}^{new}(\boldsymbol{I}) = F_{h,0}^{new} \cdot \boldsymbol{L_k}.$$

Assuming that $\boldsymbol{P_k} = [p_{k,1} \quad p_{k,2} \ \cdots \ p_{k,D_k}]^T$ and the original size of the ith dimension of X_k is $N_{k,i}$, the ith dimension of X_k needs to be expanded by $|p_{k,i}|$ units to $N_{k,i} + |p_{k,i}|$.

Fig. 11 gives an example application of global duplication. In this example, we use in-place duplication for array A. Without the use of global duplication, array B needs to be fully duplicated as shown in Fig. 11(b). In the case of global duplication, array B uses the available space in array A to store its duplicates, and Fig. 11(c) gives the transformed code. If $\mathtt{N} = 100$, by using in-place duplication, we can reduce the extra memory space from 100% (in the full duplication case) to 51%. By using global duplication, on the other hand, this number is further reduced to 2%.

5 Experimental Evaluation

5.1 Setup

In this section, we present an experimental evaluation of the approach discussed in this paper. To evaluate the effectiveness of our approach, we implemented it within an optimizing compiler [22] and performed experiments with several array based benchmarks. The average increase due to our approach in compilation times of the original

Table 2. Benchmarks used in this study

SpecFP2000		
Benchmark	**Brief Description**	**Input**
171.swim	Shallow Water Modeling	Ref. Input
172.grid	Multi-Grid Solver	Ref. Input
177.mesa	3D Graphic Library	Ref. Input
179.art	Image Recognition/Neural Networks	Ref. Input
183.equake	Seismic Wave Propagation Simulation	Ref. Input
188.ammp	Computational Chemistry	Ref. Input

Embedded Applications		
Benchmark	**Brief Description**	**Input**
atr	Network Address Translation	1.47MB
bss	Signal Deconvolution	3.07MB
encr	Digital Signature for Security	1.88MB
img-seg6	Embedded Image Segregation	2.61MB
usonic	Feature-Based Area Estimation	4.36MB
wood04	Color-Based Surface Inspection	5.28MB

programs was about 220%. Table 2 lists the benchmarks used in this study. Our benchmarks are divided into two groups. The first group contains the C benchmarks from the SpecFP2000 suite [17] (plus two FORTRAN benchmarks, of which we were able to generate the C versions by hand), whereas the second group are representative applications from the domain of embedded computing. We collected the applications in the second group from different sources. For each group of benchmarks in Table 2, the second column gives a brief description of each benchmark and the last column shows the size of the total data manipulated by each benchmark.

5.2 Results

Fig. 12 shows the effectiveness of in-place and global duplication in reducing the memory requirements due to enhanced reliability. Each bar in this figure represents the extra memory demand of the corresponding approach (in-place or global), as a fraction of the extra memory demand of a scheme that duplicates all the array data in the application (i.e., full duplication). As can be seen from this bar-chart, our approaches save significant memory space with respect to the full duplication of all array data. The average savings brought by the in-place duplication scheme are 30.2% and 36.4% for the SpecFP2000 benchmarks and the embedded applications, respectively. The corresponding savings with the global duplication scheme are 39.3% and 44.2%. We see that, except for two benchmarks (177.mesa and atr), the global duplication scheme brings savings over the in-place duplication scheme, as the former has the flexibility of using other arrays for creating duplicates of the elements of a given array.

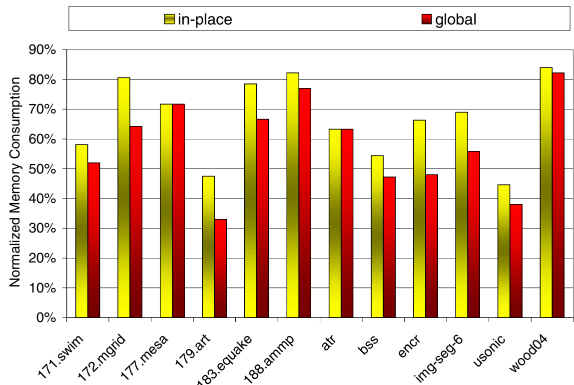


Fig. 12. Memory requirements of our duplication schemes

Note that, in the results presented above, all array elements have been duplicated (some recycling the memory locations of the elements that passed their last uses). Our next set of experiments measure the success of the in-place duplication approach that

operates with memory constraints (see Section 3.3). The results are given in Fig. 13 with different memory constraints. Specifically, each point on the x-axis gives the maximum allowable increase in the size of the data manipulated by the original programs. The y-axis, on the other hand, gives the percentage of array elements duplicated by our approach. We see from these results that, our approach is successful in utilizing available extra memory space for duplication. In fact, even with an extra 5% memory space, it is able to duplicate about 16% of the array elements on the average. When we increase the extra available memory space reserved for duplicates to 40%, the average percentage of duplication becomes 76%.

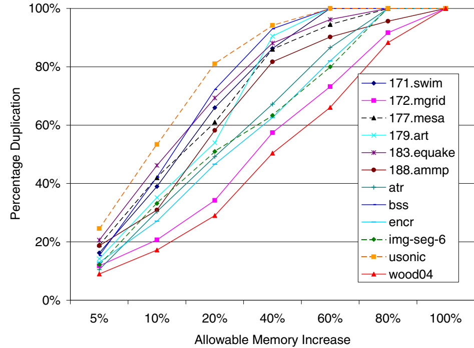


Fig. 13. Duplication under memory constraints

Although our focus in this paper is on memory space savings and reliability, it is also important to consider the impact of our approach on execution cycles. To determine the execution cycles taken by our approach, we simulated the benchmark codes using SimpleScalar [15]. The simulated architecture is a two-issue embedded processors with 16KB instruction and data caches. The access latencies for both the caches are 1 cycle, and a miss penalty of 100 cycles is assumed. The graph in Fig. 14 gives execution cycles for our two schemes as a fraction of the execution cycles taken by the full duplication strategy. Note that the full duplication scheme almost doubles the execution cycles of the original codes (i.e., those without any protection). Two observations can be made from this graph. First, both the schemes perform better than the full duplication based approach in terms of execution cycles. The main reason for this is that the reduction in data space requirements reduces capacity misses and this in turn reduces execution cycles. Second, the difference between our two schemes is less than one would expect, given the fact that global can reuse (and save) more memory space than in-place. The reason for the small difference between the two is the increased number of conflict misses with the global

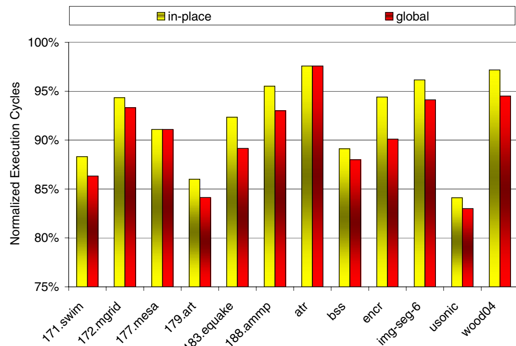


Fig. 14. Performance of our duplication schemes

scheme, due to the additional irregularity created by reusing different locations in the same loop iteration.

6 Related Work

6.1 Memory Reuse

There exist several prior studies that reduce the memory footprint of array-based programs. Wolfe [24] presented a technique called array contraction to optimize programs for a vector architecture. Lefebvre and Feautrier [8] proposed a method for reducing the memory overhead brought by full data expansion in automatic parallelization of loop-based static control programs. Song et al [16] proposed an algorithm that combines loop shifting, loop fusion, and array contraction to reduce memory usage and improve data locality. Wilde and Rajopadhye [21] studied memory reuse using a polyhedral model that performs static lifetime computations of variables. Strout et al [19] presented a schedule-independent storage mapping technique that reduces data space consumption but introduces no dependences other than those implied by flow dependences. Unnikrishnan et al [20] used a loop-based program transformation technique to reduce lifetimes of array elements. Their objective is to reduce the cases where the lifetimes of array elements overlap so that the storage requirement can be reduced. The common point between our approach and these prior studies is that all of them exploit variable lifetime information extracted by the compiler. The main difference is that we use this information for reducing the additional memory space demand due to enhanced reliability against soft errors, rather than reducing the original memory demand of the application. Also, most of these prior studies optimize for a single array at a time and operate under some additional constraints such as maintaining a certain degree of parallelism. In comparison, our global duplication scheme can reuse the space available in other arrays for storing the duplicates of the elements of a given array.

6.2 Software Approach to Transient/Permanent Errors

Software techniques for fault detection and recovery have been studied by prior research. Huang and Abraham [7] proposed Algorithm-Based Fault Tolerance (ABFT) to ensure the reliability of matrix operations. Roy-Chowdhury [13] extended the ABFT framework to a parallel processing environment. Oh and McCluskey [10] proposed Selective Procedure Call Duplication (SPCD) to improve system reliability. SPCD analyzes the procedure-call behavior of the program, and determines whether to duplicate the statements of a procedure or duplicate the procedure call. Rebaudengo et al [12] and Nicolescu et al [9] proposed systematic approaches for introducing redundancy into programs to detect errors in both data and code. Their approach demonstrated good error detection capabilities, but it also introduced considerable memory overheads due to full duplication for all variables. Our approach, in contrast, tries to minimize the memory overhead and retains the same degree of reliability that would be provided by full duplication. Audet et al [2] presented an approach for reducing a program's sensitivity to transient errors by modifying the program structure, without introducing redundancy

into the program. Although this approach introduces almost no extra memory overhead, it cannot provide the same degree of reliability that would be provided by full duplication. Benso et al [3] presented a similar work that improves the reliability of a C code by code reordering. They do not consider memory optimization through array reuse. Shirvani et al [14] used software-implemented error detection and correction (EDAC) code to provide protection against transient errors. Several prior studies targeted at specific platforms. Gong et al [5,6] proposed a compiler-assisted approach to fault detection in regular loops for distributed-memory systems. Their approach focuses on performance issues, and does not consider memory consumption. In comparison, our objective in this work is to reduce memory overheads.

7 Concluding Remarks

Many embedded systems operate under multiple constraints such as limited memory size, limited battery power, real-time performance, reliability, and security. Consequently, in optimizing for one constraint, one should be very careful in controlling the impact of doing so on other constraints. Motivated by this observation, this paper presents a memory space conscious compiler-based approach that targets improving reliability of array-based programs against soft errors, a form of transient errors. The idea is to reuse the memory locations of inactive array elements (i.e., the elements that have reached their last uses) as placeholders for the duplicates of the actively used array elements. We present two specific algorithms based on this idea, and test their effectiveness using a set of twelve array-based applications. Our experimental evaluation demonstrates that our approach is successful in reducing the extra memory demand due to improved reliability.

References

1. R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491-542, October 1987.
2. D. Audet, S. Masson, and Y. Savaria. Reducing fault sensitivity of microprocessor-based systems by modifying workload structure. In *Proc. IEEE International Symposium in Defect and Fault Tolerant in VLSI Systems*, 1998.
3. A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri. A C/C++ source-to-source compiler for dependable applications. In *Proc. International Conference on Dependable Systems and Networks*, pp. 71-78, June 2000.
4. F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. V. Achteren, and T. Omnes. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Academic Publishers, 2002.
5. C. Gong, R. Melhem and R. Gupta. Compiler assisted fault detection for distributed memory systems. In *Proc. 1994 Scalable High Performance Computing Conference*, Knoxville, TN, 1994.
6. C. Gong, R. Melhem, and R. Gupta. Loop transformations for fault detection in regular loops on massively parallel systems. *IEEE Transaction on Parallel and Distributed Systems*, 7(12):1238-1249, December 1996.
7. K. H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, vol. C-33, pp. 518-528, June 1984.

8. V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Research Report PRiSM 97/8*, France, 1997.
9. B. Nicolescu and Raoul Velazco. Detecting soft errors by a purely software approach: method, tools and experimental results. In *Proc. Design, Automation and Test in Europe Conference and Exhibition*, Munich, Germany, March 2003.
10. N. Oh and E. J. McCluskey. Error detection by selective procedure call duplication for low energy consumption. *IEEE Transactions on Reliability*, 51(4):392-402, December 2002.
11. W. Pugh, D. Wonnacott. An exact method for analysis of value-based array data dependences. In *Proc. the 6th International Workshop on Languages and Compilers for Parallel Computing*, 1993.
12. M. Rebaudengo, M. Sonza Reorda, M. Violante, P. Cheynet, B. Nicolescu, and R. Velazco. System safety through automatic high-level code transformations: an experimental evaluation. In *Proc. IEEE Design Automation and Testing in Europe*, Munich, Germany, March 13-16, 2001.
13. Amber Roy-Chowdhury. Manual and compiler assisted methods for generating error detecting parallel programs. *Ph.D thesis*, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1996.
14. P. P. Shirvani, N. Saxena, and E. J. McCluskey. Software-implemented EDAC protection against SEUs. *IEEE Transaction on Reliability*, 49(3):273-284, September 2000.
15. http://www.simplescalar.com.
16. Y. Song, R. Xu, C. Wang, and Z. Li. Data locality enhancement by memory reduction. In *Proc. the 15th ACM International Conference on Supercomputing*, June 2001.
17. http://www.spec.org/osg/cpu2000/CFP2000/.
18. G. R. Srinivasan. Modeling the cosmic-ray-induced soft-error rate in integrated circuits: an overview. *IBM Journal of Research and Development*, 40(1):77–89, January 1996.
19. M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independent storage mapping in loops. In *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.
20. P. Unnikrishnan, G. Chen, M. Kandemir, M. Karakoy, and I. Kolcu. Loop transformations for reducing data space requirements of resource-constrained applications. In *Proc. International Static Analysis Symposium*, June 11-13, 2003.
21. D. Wilde and S. Rajopadhye. Memory reuse analysis in the polyhedral model. *Parallel Processing Letters,* 1997.
22. R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31-37, December 1994.
23. M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In Proc. *ACM SIGPLAN Conference on Programming Language Design and Implementation,* pp. 30-44, June 1991.
24. M. J. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996.
25. J. F. Zeigler. Terrestrial cosmic rays. *IBM Journal of Research and Development*, 40(1):19–39, January 1996.

Memory Usage Verification for OO Programs

Wei-Ngan Chin[1,2], Huu Hai Nguyen[1], Shengchao Qin[3], and Martin Rinard[4]

[1] Computer Science Programme, Singapore-MIT Alliance
[2] Department of Computer Science, National University of Singapore
[3] Department of Computer Science, University of Durham
[4] Laboratory for Computer Science, Massachusetts Institute of Technology
{chinwn, nguyenh2}@comp.nus.edu.sg
shengchao.qin@durham.ac.uk, rinard@lcs.mit.edu

Abstract. We present a new type system for an object-oriented (OO) language that characterizes the sizes of data structures and the amount of heap memory required to successfully execute methods that operate on these data structures. Key components of this type system include type assertions that use symbolic Presburger arithmetic expressions to capture data structure sizes, the effect of methods on the data structures that they manipulate, and the amount of memory that methods allocate and deallocate. For each method, we conservatively capture the amount of memory required to execute the method as a function of the sizes of the method's inputs. The safety guarantee is that the method will never attempt to use more memory than its type expressions specify. We have implemented a type checker to verify memory usages of OO programs. Our experience is that the type system can precisely and effectively capture memory bounds for a wide range of programs.

1 Introduction

Memory management is a key concern for many applications. Over the years researchers have developed a range of memory management approaches; examples include explicit allocation and deallocation, copying garbage collection, and region-based memory allocation. However, an important aspect that has been largely ignored in past work is the safe estimation of memory space required for program execution. Overallocation of memory may cause inefficiency, while underallocation may cause software failure. In this paper, we attempt to make memory usage more predictable by static verification on the memory usage of each program.

We present a new type system, based on dependent type[21], that characterizes the amount of memory required to execute each program component. The key components of this type system include:

– **Data Structure Sizes and Size Constraints:** The type of each data structure includes index parameters to characterize its size properties, which are expressed in terms of the sizes of data structures that it contains. In many cases the sizes of these data structures are correlated; our approach uses size constraints expressed using symbolic Presburger arithmetic terms to precisely capture these correlations.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 70–86, 2005.
© Springer-Verlag Berlin Heidelberg 2005

- **Heap Recovery:** Our type system captures the distinction between shared and unaliased objects and supports explicit deallocation of unaliased objects.
- **Preconditions and Postconditions:** Each method comes with a precondition that captures both the expected sizes of the data structures on which it operates and any correlations between these sizes. The method's postcondition expresses the new size and correlations of these data structures after the method executes as a function of the original sizes when the method was invoked.
- **Heap Usage Effects:** Each method comes with two memory effects. These effects use symbolic values (present in method precondition) to capture (i) *memory requirement* which specify the maximum heap space that the method *may* consume, (ii) *memory release* which specify the minimum heap space that the method *will* recover. Heap effects are expressed at the granularity of classes and can capture the net change in the number of instances of each class.

Our paper makes several new technical contributions. Firstly, we design a formal verification system in the form of a type system, that can *formally* and *statically* capture memory usage for the object-oriented (OO) paradigm. We believe that ours is the first such formal type system for OO paradigm. Secondly, we advocate for *explicit heap recovery* to provide more timely reclamation of dead objects in support of tighter bounds on memory usage. We show how such recovery commands may be automatically inserted. Thirdly, we have proven the soundness of our type checking rules. Each well-typed program is guaranteed to meet its memory usage specification, and will *never fail due to insufficient memory* whenever its memory precondition is met. Lastly, we have implemented a type checker (with an inference mechanism) and have shown that it is fairly precise and can handle a reasonably large class of programs. Runtime **stack space** to hold methods' parameters and local variables is another aspect of memory needed. For simplicity, we omit its consideration in this paper.

2 Overview

Memory usage occurs primarily in the heap to hold dynamically created objects. In our model, heap space is consumed via the `new` operation for newly created objects, while unused objects may be recovered via an explicit deallocation primitive, called `dispose`. Memory usage (based on consumption and recovery) should be calculated over the entire computation of each program. This calculation is done in a safe manner to help identify the high watermark on memory space needed. We achieve this through the use of a conservative upper bound on memory consumed, and a conservative lower bound on memory recovered for each expression (and method).

To safely predict the memory usage of each program, we propose a *size-polymorphic type system* for object-oriented programs with support for interprocedural size analysis. In this type system, size properties of both user-defined types and primitive types are captured. In the case of primitive integer type $\texttt{int}\langle v\rangle$, the size variable v captures its integer value, while for boolean type $\texttt{bool}\langle b\rangle$, the size variable b is either 0 or 1 denoting `false` or `true`, respectively. (Note that size variables capture some integer-based properties of the data structure. For simple types, the values are directly captured.) For user-defined class types, we use $c\langle n_1,\ldots,n_p\rangle$ `where` ϕ ; ϕ_I with size variables $n_1,\ldots,n_p$ to

denote size properties that are defined in size relation ϕ, and invariant constraint ϕ_I. As an example, consider a user-defined stack class, that is implemented with a linked list, and a binary tree class as shown below.

$$
\begin{array}{l}
\texttt{class List}\langle n\rangle\ \texttt{where}\ n{=}m{+}1\ ;\ n{\geq}0\ \{\ \texttt{Object}\langle\rangle\texttt{@S val};\ \texttt{List}\langle m\rangle\texttt{@U next};\cdots\}\\
\texttt{class Stack}\langle n\rangle\ \texttt{where}\ n{=}m\ ;\ n{\geq}0\ \{\ \texttt{List}\langle m\rangle\texttt{@U head};\cdots\}\\
\texttt{class BTree}\langle s,d\rangle\ \texttt{where}\ s{=}1{+}s_1{+}s_2{\wedge}d{=}1{+}max(d_1,d_2)\ ;\ s{\geq}0{\wedge}d{\geq}0\ \{\\
\quad \texttt{Object}\langle\rangle\texttt{@S val};\ \texttt{BTree}\langle s_1,d_1\rangle\texttt{@U left};\ \texttt{BTree}\langle s_2,d_2\rangle\texttt{@U right};\cdots\}
\end{array}
$$

$\texttt{List}\langle n\rangle$ denotes a linked-list data structure of size n, and similarly for $\texttt{Stack}\langle n\rangle$. The size relations $n{=}m{+}1$ and $n{=}m$ define some size properties of the objects in terms of the sizes of their components, while the constraint $n{\geq}0$ signifies an invariant associated with the class type. Class $\texttt{BTree}\langle s,d\rangle$ represents a binary tree with size variables s and d denoting the total number of nodes and the depth of the tree, respectively. Due to the need to track the states of mutable objects, our type system requires the support of alias controls of the form $A{=}\texttt{U}\mid\texttt{S}\mid\texttt{R}\mid\texttt{L}$. We use U and S to mark each reference that is (definitely) *unaliased* and (possibly) *shared*, respectively. We use R to mark read-only fields which must never be updated after object initialization. We use L to mark unique references that are temporarily borrowed by a parameter for the duration of its method's execution. Our alias annotation mechanism are adapted from [5, 8, 1] and reported in [9]. Briefly, they allow us to track unique objects from mutable fields, as well as shareable objects from read-only fields.

To specify memory usage, we decorate each method with the following declaration:

$$t\ mn(t_1v_1,\ldots,t_nv_n)\ \texttt{where}\ \phi_{pr};\phi_{po};\epsilon_c;\epsilon_r\ \{e\}$$

where ϕ_{pr} and ϕ_{po} denote the precondition and postcondition of the method, expressed in terms of constraints/formulae on the size variables of the method's parameters and result. Precondition ϕ_{pr} denotes an applicability condition of the method in terms of the sizes of its parameters. Postcondition ϕ_{po} can provide a precise size relation for the parameters and result of the declared method. The memory effect is captured by ϵ_c and ϵ_r. Note that ϵ_c denotes *memory requirement*, i.e., the maximum memory space that *may be consumed*, while ϵ_r denotes *net release*, i.e., the minimum memory space that *will be recovered* at the end of method invocation. Memory effects (consumption and recovery) are expressed using a bag notation of the form $\{(c_i,\alpha_i)\}_{i=1}^m$, where c_i denotes a class type, while α_i denotes its symbolic count.

```
class Stack⟨n⟩ where n=m ; n≥0 { List⟨m⟩@U head;
L | void⟨⟩@S push(Object⟨⟩@S o) where true; n′=n+1; {(List, 1)}; {}
    { List⟨⟩@U tmp=this.head; this.head=new List(o, tmp)}
L | void⟨⟩@S pop() where n>0; n′=n−1; {}; {(List, 1)}
    { List⟨⟩@U t1 = this.head; List⟨⟩@U t2 = t1.next; t1.dispose(); this.head = t2}
L | bool⟨b⟩@S isEmpty() where n≥0; n′=n ∧ (n=0∧b=1 ∨ n>0∧b=0); {}; {}
    { List⟨⟩@U t = this.head; bool⟨⟩@S v = isNull(t); this.head = t; v}
L | void⟨⟩@S emptyStack() where n≥0∧d=n; n′=0; {}; {(List, d)}
    { bool⟨⟩@S v = this.isEmpty(); if v then () else {this.pop(); this.emptyStack()}}
L | void⟨⟩@S push3pop2(Object⟨⟩@S o) where true; n′=n+1; {(List, 2)}; {(List, 1)}
    { this.push(o); this.push(o); this.pop(); this.push(o); this.pop()}}
```

Fig. 1. Methods for the Stack Class

Examples of method declarations for the `Stack` class are given in Fig 1. The notation ($A \parallel$) prior to each method captures the alias annotation of the current `this` parameter. Note our use of the primed notation, advocated in [13, 17], to capture imperative changes on size properties. For the `push` method, $n'{=}n{+}1$ captures the fact that the size of the stack object has increased by 1; similarly, the postcondition for the `pop` method, $n'{=}n{-}1$, denotes that the size of the stack is decreased by 1 after the operation. The memory requirement for the `push` method, $\epsilon_r{=}\{(\texttt{List}, 1)\}$, captures the fact that one `List` node will be consumed. For the `pop` method, $\epsilon_r{=}\{(\texttt{List}, 1)\}$ indicates that one `List` node will be recovered.

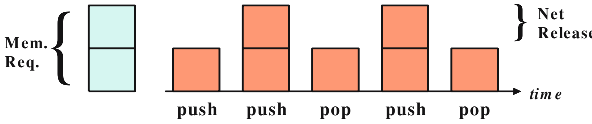


Fig. 2. `push3pop2`: Heap Consumption and Recovery

For the `isEmpty` method, $n'{=}n$ captures the fact that the size of the receiver object (`this`) is not changed by the method. Furthermore, its output of type $\texttt{bool}\langle b\rangle\texttt{@S}$ is related to the object's size through a disjunctive constraint $n{=}0{\wedge}b{=}1{\vee}n{>}0{\wedge}b{=}0$. Primitive types are annotated with alias `S` because their values are immutable and can be freely shared and yet remain trackable. The `emptyStack` method releases all `List` nodes of the `Stack` object. For `push3pop2` method, the memory consumed (or required) from the heap is $\{(\texttt{List}, 2)\}$, while the net release is $\{(\texttt{List}, 1)\}$, as illustrated in Fig. 2.

Size variables and their constraints are specified at method boundary, and need not be specified for local variables. Hence, we may use $\texttt{bool}\langle\rangle\texttt{@S}$ instead of $\texttt{bool}\langle v\rangle\texttt{@S}$ for the type of a local variable.

3 Language and Annotations

We focus on a core object-oriented language, called MEMJ, with size, alias, and memory annotations in Fig 3. MEMJ is designed to be an intermediate language for Java with either supplied or inferred annotations. A suffix notation y^* denotes a list of zero or more distinct syntactic terms that are suitably separated. For example, $(t\ v)^*$ denotes $(t_1\ v_1, \ldots, t_n\ v_n)$ where $n{\geq}0$. Local variable declarations are supported by block structure of the form: $(t\ v = e_1; e_2)$ with e_2 denoting the result. We assume a call-by-value semantics for MEMJ, where values (primitives or references) are passed as arguments to parameters of methods. For simplicity, we do not allow the parameters to be updated (or re-assigned) with different values. There is no loss of generality, as we can always copy such parameters to local variables for updating.

The MEMJ language is deliberately kept simple to facilitate the formulation of static and dynamic semantics. Typical language constructs, such as multi-declaration block, sequence, calls with complex arguments, *etc.* can be automatically translated to constructs in MEMJ. Also, loops can be viewed as syntactic abbreviations for tail-recursive methods, and are supported by our analysis. Several other language features, including downcast and a field-binding construct are also supported in our implementation. For simplicity, we omit them in this paper, as they play supporting roles and are not

$$P ::= \mathit{def}^* \ \mathit{meth}^*$$
$$\mathit{def} ::= \texttt{class}\ c_1\langle n_{1..p}\rangle\ [\texttt{extends}\ c_2\langle n_{1..q}\rangle\ ]\texttt{where}\ \phi\ ;\ \phi_I\ \{\mathit{fd}^*\ (A \parallel \mathit{meth})^*\ \}$$
$$\mathit{meth} ::= t\ \mathit{mn}((t\ v)^*)\ \texttt{where}\ \phi_{pr}; \phi_{po}; \epsilon_c; \epsilon_r\ \{e\}$$
$$\mathit{fd} ::= t\ \ f \qquad t ::= \tau\langle n^*\rangle\texttt{@}A \qquad A ::= \texttt{U}\ |\ \texttt{L}\ |\ \texttt{S}\ |\ \texttt{R}$$
$$\tau ::= c\ |\ \mathit{pr} \qquad w ::= v\ |\ v.f \qquad \mathit{pr} ::= \texttt{int}\ |\ \texttt{bool}\ |\ \texttt{void}$$
$$e ::= (c)\,\texttt{null}\ |\ k\ |\ w\ |\ w = e\ |\ t\ v = e_1\ ;\ e_2\ |\ \texttt{new}\ c\,(v^*)$$
$$|\ v.\mathit{mn}(v^*)\ |\ \mathit{mn}(v^*)\ |\ \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ |\ v.\texttt{dispose()}$$
$$\epsilon = \{(c,\alpha)^*\} \qquad \textit{(Memory Space Abstraction)}$$
$$\phi \in \mathbf{F} \qquad \textit{(Presburger Size Constraint)}$$
$$::=\ b\ |\ \phi_1 \wedge \phi_2\ |\ \phi_1 \vee \phi_2\ |\ \neg\phi\ |\ \exists n \cdot \phi\ |\ \forall n \cdot \phi$$
$$b \in \mathbf{BExp} \qquad \textit{(Boolean Expression)}$$
$$::=\ \texttt{true}\ |\ \texttt{false}\ |\ \alpha_1 = \alpha_2\ |\ \alpha_1 < \alpha_2\ |\ \alpha_1 \leq \alpha_2$$
$$\alpha \in \mathbf{AExp} \qquad \textit{(Arithmetic Expression)}$$
$$::= k^{\texttt{int}}\ |\ n\ |\ k^{\texttt{int}} * \alpha\ |\ \alpha_1 + \alpha_2\ |\ -\alpha\ |\ \mathit{max}(\alpha_1,\alpha_2)\ |\ \mathit{min}(\alpha_1,\alpha_2)$$
$$\textit{where } k^{\texttt{int}} \in Z \textit{ is an integer constant};\ n \in SV \textit{ is a size variable}$$
$$f \in Fd \textit{ is a field name};\ v \in Var \textit{ is an object variable}$$

Fig. 3. Syntax for the MEMJ Language

core to the main ideas proposed here. The interested reader may refer to our companion technical report[10] for more information.

To support sized typing, our programs are augmented with size variables and constraints. For size constraints, we restrict to Presburger form, as decidable (and practical) constraint solvers exist, e.g. [19]. We are primarily interested in tracking size properties of objects.We therefore restrict the relation ϕ in each class declaration of $c_1\langle n_1, .., n_p\rangle$ which extends $c_2\langle n_1, .., n_q\rangle$ to the form $\bigwedge_{i=q+1}^{p} n_i{=}\alpha_i$ whereby $V(\alpha_i) \cap \{n_1, .., n_p\} = \emptyset$. Note that $V(\alpha_i)$ returns the set of size variables that appeared in α_i. This restricts size properties to depend solely on the components of their objects.

Note that each class declaration has a set of instance methods whose main purpose is to manipulate objects of the declared class. For convenience, we also provide a set of static methods with the same syntax as instance methods, except for its access to the `this` object. One important feature of MEMJ is that memory recovery is done safely (without creating dangling references) through a $v.\texttt{dispose}()$ primitive.

4 Heap Usage Specification

To allow memory usage to be precisely specified, we propose a bag abstraction of the form $\{(c_i, \alpha_i)\}_{i=1}^{n}$ where c_i denotes its classification, while α_i is its cardinality. In this paper, we shall use $c_i \in CN$ where CN denotes all class types. For instance, $\Upsilon_1 = \{(c_1, 2), (c_2, 4), (c_3, \texttt{x} + 3)\}$ denotes a bag with c_1 occurring twice, c_2 four times and c_3 $\texttt{x} + 3$ times. We provide the following two basic operations for bag abstraction to capture both the domain and the count of its element, as follows:

$$dom(\Upsilon) =_{df} \{c \mid (c,n) \in \Upsilon\} \qquad \Upsilon(c) =_{df} \begin{cases} n, & \textit{if } (c,n) \in \Upsilon \\ 0, & \textit{otherwise} \end{cases}$$

We define union, difference, exclusion over bags as:

$$\begin{array}{lll} \Upsilon_1 \uplus \Upsilon_2 & =_{df} & \{(c, \Upsilon_1(c){+}\Upsilon_2(c)) \mid c \in dom(\Upsilon_1) \cup dom(\Upsilon_2)\} \\ \Upsilon_1 - \Upsilon_2 & =_{df} & \{(c, \Upsilon_1(c){-}\Upsilon_2(c)) \mid c \in dom(\Upsilon_1) \cup dom(\Upsilon_2)\} \\ \Upsilon \setminus X & =_{df} & \{(c, \Upsilon(c)) \mid c \in dom(\Upsilon) - X\} \end{array}$$

To check for adequacy of memory, we provide a bag comparator operation under a size constraint Δ, as follows:

$$\Delta \vdash \Upsilon_1 \sqsupseteq \Upsilon_2 =_{df} (\Delta \Rightarrow (\forall c \in Z \cdot \Upsilon_1(c) \geq \Upsilon_2(c))) \ \texttt{where}\ Z = dom(\Upsilon_1) \cup dom(\Upsilon_2)$$

The bag abstraction notation for memory is quite general and can be made more precise by refining its operations. For example, some class types are of the same size and could replace each other to increase memory reuse. To achieve this we can use a bag abstraction that is grouped by $size(c_i)$ instead of class type c_i.

4.1 Heap Consumption

Heap space is consumed when objects are created by the `new` primitive, and also by method calls, except that the latter is aggregated to include recovery prior to consumption. Our aggregation (of recovery prior to consumption) is designed to identify a high watermark of maximum memory needed for safe program execution. For each expression, we predict a conservative upper bound on the memory that the expression *may* consume, and also a conservative lower bound on the memory that the expression *will* release. If the expression releases some memory before consumption, we will use the released memory to obtain a lower memory requirement. Such aggregated calculations on both consumption and recovery can help capture both a net change in the level of memory, as well as the high watermark of memory needed for safe execution.

For example, consider a recursive function which does p pops from one stack object, followed by the same number of pushes on another stack.

```
void⟨⟩@S moverec(Stack⟨a⟩@L s, Stack⟨b⟩@L t, int⟨p⟩@S i)
   where a≥p≥0; a′=a−p∧b′=b+p; {} ; {}
 { if i<1 then ()
   else {Object⟨⟩@S o = s.top(); s.pop(); moverec(s, t, i−1); t.push(o)} }
```

Due to aggregation (involving recovery before consumption), the heap space that may be consumed is zero. For each recursive call, the space for a `List` node is released by `s.pop()` before it is reused by `t.push(o)`. Aggregated over the recursive calls, we will have p number of `List` nodes that have been released before the same number of nodes are consumed. Hence, no new heap space is needed. Such aggregation is sensitive to the order of the operations.

Consider now a different function which performs p pushes on `t`, followed by the same number of pops from `s`.

```
void⟨⟩@S moverec2(Stack⟨a⟩@L s, Stack⟨b⟩@L t, int⟨p⟩@S i)
   where a≥p≥0; a′=a−p∧b′=b+p; {(List, p)}; {(List, p)}
 { if i<1 then ()
   else {Object⟨⟩@S o = s.top(); t.push(o); moverec2(s, t, i−1); s.pop()} }
```

Though the net change in memory usage is also zero, the memory effect for this function is different as we require p number of `List` nodes to be consumed on entry, *before* the same number of `List` nodes are recovered. This new memory effect has the potential to push up the high watermark of memory needed by p `List` nodes.

4.2 Heap Recovery

Explicit heap space recovery via `dispose` has several advantages. It facilitates the timely recovery of dead objects, which allows memory usage to be predicted more accurately (with tighter bounds). It also permits the use of more efficient custom allocators[4], where desired. Moreover, we shall provide an automatic technique to insert `dispose` primitives with the help of alias annotation. With such a technique, we only need to ensure that objects that are being disposed are non-null. This non-nullness property can be captured by a non-nullness analyser, such as [12]. This property is required as we always recover memory space for each `dispose` primitive.

Memory recovery via `dispose` should occur when unique references that are still alive (not in dead-set) are being discarded. This could occur at four places[1] : (i) end of local block, (ii) end of method block, (iii) prior to assignment operation, and (iv) at conditional expression. We would like to recover the memory space for each non-null reference that is about to become dead. For example, consider the `pop` method's definition:

$$\mathtt{L} \mid \mathtt{void}\langle\rangle\mathtt{@S}\ \mathtt{pop}()\ \mathtt{where}\ \cdots\ \{\ \mathtt{List}\langle\rangle\mathtt{@U}\ \mathtt{t1} = \mathtt{this.head};\ \mathtt{head} = \mathtt{t1.next}\}$$

The object pointed to by `head` is about to become dead prior to the operation, $\mathtt{head} = \mathtt{t1.next}$. To recover this dead object, we insert a `dispose` command to obtain $\mathtt{head} = (\mathtt{t1.next} <; \mathtt{head.dispose}())$ where $e_1 {<;} e_2 \equiv (t\ v = e_1; e_2; v)$. Consider the definition of the `destroy` method which calls `emptyStack` with an L-mode parameter.

$$\mathtt{void}\langle\rangle\mathtt{@S}\ \mathtt{destroy}(\mathtt{Stack}\langle n\rangle\mathtt{@U}\ \mathtt{s})\ \mathtt{where}\ \cdots\{\mathtt{emptyStack(s)}\}$$

A unique `s` object is about to become dead at the end of the `destroy` method. To recover this space, we can insert `s.dispose()` prior to the method's exit.

Let us formalise an automatic technique for the explicit recovery of dead objects that are known at compile-time. Given an expression e, we utilize the alias annotation to obtain a new expression e_1 where suitable explicit heap `dispose` operations have been safely inserted. This is achieved by a translation below with Γ to denote a type environment mapping program variables to their annotated types, and $\Theta(\Theta_1)$ to denote the set of dead references (of the form v or $v.f$) before (after) the evaluation of expression e.

$$\Gamma; \Theta \vdash e \hookrightarrow_H e_1 :: t, \Theta_1$$

Most rules are structure-preserving (or identity) rewritings, except for four rules given in Fig 4. A sequence of disposals can be effected through $\mathit{dispose}(D)$, with D containing a set of variable/field references that are about to be dead at the end of expression e.

For the assignment rule [H:ASSIGN], we add w to the disposal set if it is unique and is not yet in dead-set using $D = \{w \mid \mathit{ann}(t){=}\mathtt{U}\} - \Theta_1$. The function $\mathit{isParam}(w)$ returns

[1] Note that unique reference cannot escape through e_1 in $e_1; e_2$ as we require e_1 to be of the `void` type.

$$\frac{\begin{array}{c}\neg\,\mathit{isParam}(w)\quad \Gamma(w)=t\\ D=\{w\mid \mathit{ann}(t)=\mathtt{U}\}-\Theta_1\\ \Gamma;\Theta\vdash e\hookrightarrow_H e_1::t_1,\Theta_1\\ \vdash t_1<:t\\ e_2=(e_1\lhd D{=}\emptyset\rhd e_1{<};\mathit{dispose}(D))\end{array}}{\begin{array}{l}\Gamma;\Theta\vdash w=e\hookrightarrow_H\\ \qquad w=e_2::\mathtt{void@S},\Theta_1\backslash w\end{array}}\;[\textsc{h:assign}]$$

$$\frac{\begin{array}{c}\Gamma(v)=\mathtt{bool}\langle b\rangle\mathtt{@S}\\ \Gamma;\Theta\vdash e_i\hookrightarrow_H \hat{e}_i::t_i,\Theta_i\quad i=1,2\\ t=\mathit{msst}(t_1,t_2)\quad \Theta_3=\Theta_1\cup\Theta_2\\ D_i=\Theta_3-\Theta_i\quad i=1,2\\ E_i=(\hat{e}_i\lhd D_i{=}\emptyset\rhd \hat{e}_i{<};\mathit{dispose}(D))\quad i=1,2\end{array}}{\begin{array}{l}\Gamma;\Theta\vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\hookrightarrow_H\\ \qquad \mathtt{if}\ v\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2::t,\Theta_3\end{array}}\;[\textsc{h:if}]$$

$$\frac{\begin{array}{c}\Gamma_1=\Gamma+\{v_1::t_1,..,v_p::t_p\}\\ \Gamma_1;\emptyset\vdash e\hookrightarrow_H e_1::t,\Theta\\ \vdash t<:t_0\quad \mathit{ann}(t_0)\neq\mathtt{L}\\ \forall i{\in}1..p\cdot(\mathit{ann}(t_i){=}\mathtt{L})\Rightarrow(\forall f\cdot v_i.f{\notin}\Theta)\\ D=\{w\mid(w::t)\in\Gamma_1,\mathit{ann}(t){=}\mathtt{U}\}{-}\Theta\\ e_2=(e_1\lhd D{=}\emptyset\rhd e_1{<};\mathit{dispose}(D))\end{array}}{\begin{array}{l}\Gamma\ \vdash_{meth} t_0\ mn((t_i\ v_i)_{i:1..p})\{e\}\\ \qquad\hookrightarrow_H t_0\ mn((t_i\ v_i)_{i:1..p})\ \{e_2\}\end{array}}\;[\textsc{h:meth}]$$

$$\frac{\begin{array}{c}\Gamma;\Theta\vdash e_1\hookrightarrow_H e_3::t_1,\Theta_1\\ \vdash t_1<:t\\ \mathit{ann}(t)\notin\{\mathtt{L},\mathtt{R}\}\\ \Gamma{+}\{v::t\};\Theta_1\vdash e_2\hookrightarrow_H e_4::t_2,\Theta_2\\ D=\{v\mid \mathit{ann}(t)=\mathtt{U}\}-\Theta_2\\ e_5=(e_4\lhd D{=}\emptyset\rhd e_4{<};\mathit{dispose}(D))\end{array}}{\begin{array}{l}\Gamma;\Theta\vdash(t\ v=e_1\,\mathbf{;}\,e_2)\hookrightarrow_H\\ \qquad (t\ v=e_3\,\mathbf{;}\,e_5)::t_2,\Theta_2\backslash v\end{array}}\;[\textsc{h:local}]$$

Fig. 4. Automatic Insertion of `dispose` operation

`true` if w is a parameter variable, otherwise it returns `false` (for fields and local variables). The function *ann* extracts the alias of an annotated type, $\mathit{ann}(\tau\langle v^*\rangle\mathtt{@}A)=A$. A conditional is expressed as $\xi_1\lhd b\rhd\xi_2\ =_{df}\ \begin{cases}\xi_1, & \mathit{if}\,b;\\ \xi_2, & \mathit{otherwise}.\end{cases}$ Furthermore, we have:

$$\Theta\backslash v=_{df}\Theta-\{v,v.f^*\}\qquad \Theta\backslash v.f=_{df}\Theta-\{v.f\}$$

For the method declaration rule [H:METH], we add to the disposal set those parameters which are unique but not yet dead using $\{w\mid(w::t)\in\Gamma_1,\mathit{ann}(t)=\mathtt{U}\}-\Theta$. For the local declaration rule [H:LOCAL], we add v to the disposal set if it is unique but not yet dead using $\{v\mid\mathit{ann}(t)=\mathtt{U}\}-\Theta_2$. For the [H:IF] rule, the uniqueness that are consumed in one branch may have their heap spaces recovered in the other branch. This is captured by $D_i=\Theta_3{-}\Theta_i$, $i=1,2$. Notice that $\mathit{msst}(t_1,t_2)$ returns the minimal supertype of both t_1 and t_2, as follows:

$$\frac{\begin{array}{c}\tau_1<:\tau\quad \tau_2<:\tau\quad \forall\tau_3\cdot(\tau_1,\tau_2<:\tau_3{\Rightarrow}\tau<:\tau_3)\\ A_1{\leq_a}A\quad A_2{\leq_a}A\quad \forall A_3\cdot(A_1,A_2{\leq_a}A_3{\Rightarrow}A{\leq_a}A_3)\end{array}}{\mathit{msst}(\tau_1\mathtt{@}A_1,\tau_2\mathtt{@}A_2)=_{df}\ \tau\mathtt{@}A}$$

Note that $\tau_1<:\tau_2$ denotes the subtype relation for underlying types (without annotations). Alias subtyping rules (shown below) allow unique references to be passed to shared and lent-once locations (in addition to other unique locations), but not vice-versa.

$$A\leq_a A\qquad \mathtt{U}\leq_a\mathtt{L}\qquad \mathtt{U}\leq_a\mathtt{S}$$

In the rest of this paper, we shall present a new static type system for verifying memory heap usage, followed by a set of safety theorems on the type rules.

5 Rules for Memory Checking

We present type judgements for *expressions*, *method declarations*, *class declarations* and *programs* to check for adequacy of memory, using relations of the form:

$$\Gamma;\Delta;\Upsilon \vdash e :: t, \Delta_1, \Upsilon_1 \qquad \Gamma \vdash_{meth} meth \qquad \vdash_{class} def \qquad \vdash P$$

Note that Γ is the type environment as explained earlier; $\Delta(\Delta_1)$ denotes the size constraint, which holds for the size variables associated with Γ (Γ and t) for expression e before (after) its evaluation; t is an annotated type. Also, $\Upsilon(\Upsilon_1)$ is used to denote the available memory space in terms of bag abstraction before (after) the evaluation.

We present a few key syntax-directed type rules in Fig 5, with the rest of the rules in the technical report. Before that, let us describe some notations used by the type rules.

[ASSIGN]

$$\frac{\Gamma;\Delta;\Upsilon \vdash e :: t_1, \Delta_1, \Upsilon_1 \quad \Gamma \vdash w :: t, \phi, Y \qquad \vdash t_1 {<:} t, \rho \quad X{=}V(t_1){\cup}V(t) \quad \Delta_2{=}\exists X{\cdot}(\Delta_1 \circ_Y \rho\phi)}{\Gamma;\Delta;\Upsilon \vdash w = e :: \texttt{void}\langle\rangle\texttt{@S}, \Delta_2, \Upsilon_1}$$

[DISPOSE]

$$\frac{\Gamma(v) = c\langle n^*\rangle\texttt{@U} \quad \Upsilon_1 = \Upsilon \uplus \{(c,1)\}}{\Gamma;\Delta;\Upsilon \vdash v.\texttt{dispose}() :: \texttt{void}\langle\rangle\texttt{@S}, \Delta, \Upsilon_1}$$

[NEW]

$$\frac{\begin{array}{c} \mathit{fdList}(c\langle n^*\rangle) = ([(\hat{t}_i\ f_i)]_{i=1}^{p},\ \phi') \\ r^* = \mathit{fresh}() \quad t_i = \mathit{prime}(\Gamma(v_i)) \\ \vdash t_i <: [\texttt{R} \mapsto \texttt{S}]\hat{t}_i, \rho_i\ i{\in}1..p \\ \rho = [n^* \mapsto r^*] \cup \bigcup_{i=1}^{p}\rho_i \\ \Delta \vdash \Upsilon \sqsupseteq \{(c,1)\} \quad X = \bigcup_{i=1}^{p} V(\hat{t}_i) \\ \Delta_1 = \Delta{\wedge}(\exists X{\cdot}\rho\phi') \quad \Upsilon_1 = \Upsilon{-}\{(c,1)\} \end{array}}{\Gamma;\Delta;\Upsilon \vdash \texttt{new}\ c(v_{1..p}) :: c\langle r^*\rangle\texttt{@U}, \Delta_1, \Upsilon_1}$$

[IF]

$$\frac{\begin{array}{c} \Gamma(v) = \texttt{bool}\langle b\rangle\texttt{@S} \\ \Gamma;\Delta \wedge b' = 1;\Upsilon \vdash e_1 :: t_1, \Delta_1, \Upsilon_1 \\ \Gamma;\Delta \wedge b' = 0;\Upsilon \vdash e_2 :: t_2, \Delta_2, \Upsilon_2 \\ (t, \Upsilon_3, \Delta_3) = \mathit{unify}(t_1, t_2, \Upsilon_1, \Upsilon_2, \Delta_1, \Delta_2) \end{array}}{\Gamma;\Delta;\Upsilon \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 :: t, \Delta_3, \Upsilon_3}$$

[OVERRIDE]

$$\frac{\begin{array}{c} \mathit{meth}_k = t\ mn((t_i\ v_i)_{i:1..p})\ \texttt{where} \\ \phi_{pr_k}; \phi_{po_k}; \epsilon_{km}; \epsilon_{kn}\ \{\cdots\},\ k = 1,2 \\ \phi_{pr_1} {\Rightarrow} \phi_{pr_2} \quad \phi_{po_2} {\Rightarrow} \phi_{po_1} \\ \phi_{pr_1} \vdash \epsilon_{1m} {\sqsupseteq} \epsilon_{2m} \quad \phi_{pr_1} \vdash \epsilon_{2n} {\sqsupseteq} \epsilon_{1n} \end{array}}{\vdash \mathit{OverridesOK}(\mathit{meth}_1, \mathit{meth}_2)}$$

[IMI]

$$\frac{\begin{array}{c} \vdash (A \,\|\, \hat{t}\ mn((\hat{t}_i\ \hat{v}_i)_{i:1..p})\ \texttt{where}\ \phi_{pr}; \phi_{po}; \epsilon_c; \epsilon_r\{e\}) {\in} c\langle n^*\rangle \\ t = \mathit{fresh}(\hat{t}) \quad t_0 = c\langle n^*\rangle\texttt{@}A \quad \Gamma(v_i) = t_i \ \ i{\in}0..p \quad \vdash t_i <: \hat{t}_i,\ \rho_i \ \ i{\in}1..p \\ \rho_p = \bigcup_{i=1}^{p} \rho_i \quad \Delta_1 \vdash \Upsilon {\sqsupseteq} \epsilon_c \quad \rho{=}\mathit{rename}(\hat{t}, t){\cup}\rho_p{\cup}\mathit{prime}(\rho_p) \\ \Delta \approx\!\!>_{V(\Gamma)} \exists V(\epsilon_c){\cup}V(\epsilon_r){\cdot}\rho\ \phi_{pr} \quad \Delta_1 = \Delta \circ_L \exists Y \cdot \rho(\phi_{pr}{\wedge}\phi_{po}) \\ \Upsilon_1 = \Upsilon{-}\epsilon_c{\uplus}\epsilon_r \quad X = \bigcup_{i=1}^{p} V(\hat{t}_i) \quad Y = X \cup \mathit{prime}(X) \quad L = \bigcup_{i=0}^{p} V(t_i) \end{array}}{\Gamma;\Delta;\Upsilon \vdash v_0.mn(v_{1..p}) :: t, \Delta_1, \Upsilon_1}$$

[METH]

$$\frac{\begin{array}{c} \Gamma_1 = \Gamma \cup \{v_1 :: \hat{t}_1, .., v_p :: \hat{t}_p\} \quad \Delta = \mathit{no}\mathcal{X}(\Gamma_1){\wedge}\phi_{pr}{\wedge}\mathit{inv}(\Gamma_1) \quad \Delta{\vdash}\epsilon_c{\sqsupseteq}\emptyset \\ \Gamma_1;\Delta;\epsilon_c \vdash e :: t, \Delta_1, \Upsilon_1 \quad \phi_{pr}{\wedge}\Delta_1{\vdash}\Upsilon_1 \sqsupseteq \epsilon_r \quad \Delta{\vdash}\epsilon_r{\sqsupseteq}\emptyset \quad \vdash t <: \hat{t}, \rho \\ (_, _, N_i) = \mathbf{V}_{\mathit{field}}(\hat{t}_i), i{\in}1..p \quad Y = \bigcup_{i=1}^{p} N_i \quad (\exists\, \mathit{prime}(Y){\cdot}\Delta_1){\Rightarrow}\rho(\phi_{po}) \end{array}}{\Gamma \vdash_{meth} \hat{t}\ mn((\hat{t}_i\ v_i)_{i:1..p})\ \texttt{where}\ \phi_{pr}; \phi_{po}; \epsilon_c; \epsilon_r\ \{e\}}$$

Fig. 5. Some Type Rules for Memory Checking

5.1 Notations

We use function V to return size variables of a formula, e.g. $V(x'{=}z{+}1{\wedge}y{=}2){=}\{x', y, z\}$. We extend it to annotated type, type environment, and memory specification, e.g., $V(\tau\langle n^*\rangle @A){=}\{n^*\}$, $V(\{(c, 4{\times}d{+}8)\}){=}\{d\}$. The function *prime* takes a set of size variables and returns their primed version, e.g. $prime(\{s_1, \ldots, s_n\}){=}\{s'_1, \ldots, s'_n\}$. Note that prime operation is idempotent, namely $(v')'{=}v'$. We extend this to (annotated) type, type environment,and even substitution. For example, $prime(\tau\langle n_1, \ldots, n_k\rangle) = \tau\langle n'_1, \ldots, n'_k\rangle$, and $prime([x{\mapsto}a, y{\mapsto}b]) = [x'{\mapsto}a', y'{\mapsto}b']$. Often, we need to express a no-change condition on a set of size variables. We define a $no\mathcal{X}$ operation as follows which returns a formula for which the original and primed variables are made equal.

$$no\mathcal{X}(\{\}) =_{df} \texttt{true} \qquad no\mathcal{X}(\{x\}{\cup}X) =_{df} (x'{=}x){\wedge}no\mathcal{X}(X)$$

We extend this function to annotated types (and type environments), as follows: $no\mathcal{X}(t) =_{df} no\mathcal{X}(V(t))$. Also, we use $n^* = fresh()$ to generate new size variables n^*. We extend it to annotated type, so that $\hat{t} = fresh(t)$ will return a new type $\hat{t}$ with the same underlying type as t but with fresh size variables instead. Function $rename(t_1, t_2)$ returns an equality substitution, e.g. $rename(\texttt{Int}\langle r\rangle, \texttt{Int}\langle s'\rangle){=}[r{\mapsto}s']$. The operator $\cup$ combines two domain disjoint substitutions into one.

The function *fdList* is used to retrieve a full list of fields for a given class, together with its size relation. The function *inv* is used to retrieve the size invariant that is associated with each type. This function shall also be extended to type environment and list of types. The function $\mathbf{V}_{\mathit{field}}$ classifies size variables from each field into three groups : (i) immutable, (ii) mutable but unique, (iii) otherwise (non-trackable).

To effect a change ϕ to an existing poststate Δ, we provide an operator, $\circ_Y$, with $Y = \{s^*\}$ to denote the set of size variables that is to be updated, as follows:

$$\Delta \circ_Y \phi =_{df} \exists\, r_1 \cdots r_n \cdot \rho_2(\Delta) \wedge \rho_1(\phi)$$
$$\text{where } Y = \{s_1, \ldots, s_n\}\ ;\ \{r_1, \ldots, r_n\} = fresh()\ ;\ \rho_1 = [s_i \mapsto r_i]_{i=1}^n\ ;\ \rho_2 = [s'_i \mapsto r_i]_{i=1}^n$$

5.2 Assignment

The [ASSIGN] rule captures imperative updates (to object fields and variables) by modifying the current size constraint to a new updated state with changes to the imperative size variables from the LHS. From the rule, note that $\Gamma \vdash w :: t, \phi, Y$ is to identify Y as a set of imperative size variables and also to gather a constraint ϕ for this set. The subtype relation $\vdash t_1 <: t, \rho$ will return a substitution that maps the size variables of supertype to that of the subtype. This mapping ignores all non-trackable size variables that may be globally aliased, but immutable and unique mutable size variables are captured.

5.3 Memory Operations

The heap space is directly changed by the `new` and `dispose` primitives. Their corresponding type rules, [NEW] and [DISPOSE], would ensure that sufficient memory is available for consumption by `new` and will credit back space relinquished by `dispose`. The memory effect is accumulated according to the flow of computation. Consider:

$$\cfrac{\cfrac{\Delta \vdash \Upsilon \sqsupseteq \{(\mathtt{List},1)\} \quad \Delta_1 = \Delta \circ_{\{x\}} x' = x+1}{\Gamma;\Delta;\Upsilon \vdash \mathtt{x} = \mathtt{new\ List(o,x)} :: \mathtt{void}\langle\rangle\mathtt{@S}, \Delta_1, \Upsilon - \{(\mathtt{List},1)\}} \quad \cfrac{\Upsilon_1 = (\Upsilon - \{(\mathtt{List},1)\}) \uplus \{(\mathtt{List},1)\}}{\Gamma;\Delta_1;\Upsilon - \{(\mathtt{List},1)\} \vdash \mathtt{y.dispose()} :: \mathtt{void}\langle\rangle\mathtt{@S}, \Delta_1, \Upsilon_1}}{\Gamma;\Delta;\Upsilon \vdash \mathtt{x} = \mathtt{new\ List(o,x);\ y.dispose()} :: \mathtt{void}\langle\rangle\mathtt{@S}, \Delta_1, \Upsilon}$$

The `new` operation consumes a `List` node, while the `dispose` operation releases back a `List` node. The net effect is that available memory Υ is unchanged. However, due to the order of the two operations, we require $\Delta \vdash \Upsilon \sqsupseteq \{(\mathtt{List}, 1)\}$ which affects the maximum memory required.

Another rule which has a direct effect on memory is the method invocation rule [IMI]. Sufficient memory must be available for consumption prior to each call (as specified by $\Delta_1 \vdash \Upsilon \sqsupseteq \epsilon_c$), with the net memory release added back in the end (as specified by $\Upsilon_1 = \Upsilon - \epsilon_c \uplus \epsilon_r$). Each method precondition must be met by the pre-state of its caller. This is checked by $\Delta \approx\!> _{V(\Gamma)} \exists V(\epsilon_c) \cup V(\epsilon_r) \cdot \rho\ \phi_{pr}$ which uses a relation $\approx\!>_X$, defined as:

$$\Delta \approx\!>_X \phi \ =_{df} \ (\Delta \Rightarrow \rho\phi), \text{ where } \rho = [s_1 \mapsto s_1', .., s_n \mapsto s_n'] \ \wedge \ V_u(\phi) \cap X = \{s_1, .., s_n\}.$$

Note that V_u returns size variables in unprimed form, e.g. $V_u(x'=z+1 \wedge y=2) = \{x, y, z\}$.

5.4 Conditional

Our type rule for conditional [IF] is able to track both the size-constraints and memory usages in a path-sensitive manner. Path-sensitivity is encoded by adding $b'=1$ and $b'=0$ to the pre-states of the two branches, respectively. We achieve path-sensitivity for memory usage specification by integrating it with relational size constraints derived. Take note that the *unify* operation merges the post-state constraints and memory usages from the two branches via a disjunction, a formal definition and an example can be found in our report [10]. Path-sensitivity makes our analysis more accurate and is critical for analysing the memory requirement of recursive methods.

5.5 Method Declaration

Each method declaration is checked to see if its definition is consistent with the memory usage specification given in its declaration header by the [METH] rule. The initial memory is ϵ_c. The final available memory of the method body e is Υ_1 which must not be less than the declared net memory release (as specified by $\phi_{pr} \wedge \Delta_1 \vdash \Upsilon_1 \sqsupseteq \epsilon_r$).

Function subtyping for the OO paradigm is used to support method overriding. This is captured by the [OVERRIDE] rule in Fig 5. Each method which overrides another is expected to be *contravariant* on its precondition (and memory consumption) and *covariant* on its postcondition (and memory releases).

6 Soundness of Type System

We have proposed a small-step operational semantics (denoted by $\hookrightarrow$ transitions) instrumented with alias and size notations[10], and have also formalised two safety theorems

for our type rules. The first theorem states that each well-typed expression preserves its type under reduction with a runtime environment Π and a store ϖ that are consistent with the compile-time counterparts, Γ (type environment) and Σ (store typing). Also, final size constraint is consistent with the value obtained on termination.

Theorem 1 (Preservation).

(a) (Expression) If $\Gamma;\Sigma;\Delta;\Theta;\Upsilon \vdash e :: t, \Delta_1, \Theta_1, \Upsilon_1 \quad \Gamma;\Sigma;\Delta;\Theta;\Upsilon \models \langle \Pi, \varpi, \sigma \rangle$

$$\langle \Pi, \varpi, \sigma \rangle\ [e] \hookrightarrow \langle \Pi_1, \varpi_1, \sigma_1 \rangle\ [e_1]$$

then there exist $\Sigma_\alpha \supseteq \Sigma$, Γ_α, Δ_α, Θ_α, *and* Υ_α, *such that*

$$\Gamma - \mathit{diff}(e, e_1) = \Gamma_\alpha - \mathit{diff}(e_1, e) \qquad \Gamma_\alpha;\Sigma_\alpha;\Delta_\alpha;\Theta_\alpha;\Upsilon_\alpha \vdash e_1 :: t, \Delta_1, \Theta_1, \Upsilon_1$$
$$\Gamma_\alpha;\Sigma_\alpha;\Delta_\alpha;\Theta_\alpha;\Upsilon_\alpha \models \langle \Pi_1, \varpi_1, \sigma_1 \rangle\ .$$

(b) (Value) If $\Gamma;\Sigma;\Delta;\Theta;\Upsilon \vdash (A, \delta) :: t, \Delta_1, \Theta_1; \Upsilon_1 \quad \Gamma;\Sigma;\Delta;\Theta;\Upsilon \models \langle \Pi, \varpi, \sigma \rangle$
then the following hold:

$$\Theta = \Theta_1 \qquad \Gamma + \{x :: t\};\Sigma;\Delta_2;\Theta_1;\Upsilon_1 \models \langle \Pi + \{x \mapsto (A, \delta)\}, \varpi, \sigma \rangle$$

where $x = \mathit{fresh}()$, $\Delta_2 = [v \mapsto v']_{v \in V(t)} \Delta_1$.

Proof: By induction over the depth of type derivation for expression e. Details are given in the technical report [10]. □

The second safety theorem on progress captures the fact that well-typed programs cannot go wrong. Specifically, this theorem guarantees that no memory adequacy errors are ever encountered for well-typed MEMJ programs, as follows:

Theorem 2 (Progress). *If* $\Gamma;\Sigma;\Delta;\Theta;\Upsilon \vdash e :: t, \Delta_1, \Theta_1, \Upsilon_1$ *and* $\Gamma;\Sigma;\Delta;\Theta;\Upsilon \models \langle \Pi, \varpi, \sigma \rangle$, *then either* e *is a value, or* $\langle \Pi, \varpi, \sigma \rangle\ [e] \hookrightarrow$ **Err-Null**, *or there exist* $\Pi_1, \varpi_1, \sigma_1, e_1$ *such that* $\langle \Pi, \varpi, \sigma \rangle\ [e] \hookrightarrow \langle \Pi_1, \varpi_1, \sigma_1 \rangle\ [e_1]$.

Proof: By induction over the depth of type derivation for expression e. Details are given in the technical report [10]. □

7 Implementation

We have constructed a type checker for MEMJ, and have also built a preprocessor to allow a more expressive language to be accepted. The entire prototype was built using a Haskell compiler[18] where we have added a library (based on [19]) for Presburger arithmetic constraint-solving.

The main objective of our initial experiments is to show that our memory usage specification mechanism is expressive and that such an advanced form of type checking is viable. We converted to MEMJ a set of programs from the Java version of the Olden benchmark suite [7] and another set of smaller programs from the RegJava benchmark[11], before subjecting them to memory adequacy checking. Our initial experimental results are encouraging; however this is a proof-of-concept implementation and there is scope for optimization and more exhaustive experimentation.

Programs	Size (lines)		Checking (in sec.)		Verified
	Source	Ann.	Alias	Memory	Methods
bisort	340	7	0.01	2.56	6/6
em3d	462	19	0.05	1.14	20/20
health	562	22	0.05	6.37	15/15
mst	473	31	0.02	1.26	22/22
power	765	24	0.06	4.28	19/19
treeadd	195	6	0.02	0.32	4/4
tsp	545	10	0.02	3.54	9/9
perimeter	745	12	0.02	21.81	8/8
n-body	1128	31	0.60	1.25	22/22
Voronoi	1000	45	0.03	3.51	39/40
stack	122	12	0.01	0.08	10/10
sieve	88	7	0.01	0.09	6/6
m-sort	183	13	0.01	0.36	12/12
life	164	9	0.02	2.95	7/7
Mandelbrot	194	11	0.01	1.72	10/10
Reynolds3	98	6	0.01	0.18	4/4

Fig. 6. Type Checking Experimental Results

Figure 6 summarises the statistics obtained for each program that we have verified via our type checker. Column 3 illustrates the size and memory annotation overheads which must be made in the header declarations of each class and method. Columns 4 and 5 highlight the CPU times used (in seconds) for alias and memory checking, respectively. Our experiments were done under Redhat Linux 9.0 platform on Pentium 2.4 GHz with 768MB main memory. Except for the perimeter program (which has more conditionals from using a quadtree data structure), all programs take under 10 seconds to verify, despite them being medium-sized programs and the high complexity of Presburger solving. We attribute this to the fact that memory declarations are verified in a summary-based fashion for each method definition.The last column highlights the number of methods that have been successfully verified as using memory spaces that are bounded by symbolic Presburger formulae.All methods' heap usage could be statically bounded, except[2] for a method in Voronoi that has an allocation inside a loop, with a complex termination condition.

Program	Input Size	Prediction (a)	Actual (b)	Allocation (c)	Reuse (b/c)	Accuracy (b/a)
sieve	10000	10000	9999	10000	0.9999	0.9999
m-sort	10000	20000	20000	287232	0.0696	1.0000
life	1000	2	2	1000	0.0020	1.0000
Mandelbrot	100	4	4	83692	0.00005	1.0000
Reynolds	10000	20014	20014	40000	0.5004	1.0000

Fig. 7. Experimental Results on Memory Prediction and Recovery

We have also conducted a set of experimental results to evaluate on the effectiveness of memory inference, in conjunction with our explicit memory recovery scheme. We modified IBM's Jikes RVM[2, 16] to provide support for explicit *dispose* operation and instrumented its memory system to capture total allocation (c) and actual high watermark (b). We then compare it against the predicted memory requirement (a) from our memory inference. We count the number of objects created and reused. As can be seen in Fig 7, our memory inference is accurate for the RegJava benchmark. Except for *sieve*,

[2] For Olden programs which built tree-like data structure, we make a minor change to take total nodes rather than heights as parameters. This avoids exponential formulae.

most of the programs have high degree of memory reuse which were facilitated by our use of the *dispose* operation for memory recovery.

8 Related Work

Past research on memory models for object-oriented paradigm have focused largely on efficiencyand safety. We are unaware of any prior type-based work on analysing heap memory usage by OO programs for the purpose of checking for memory adequacy. The closest related work on memory adequacy are based on first-order functional paradigm, where data structures are mostly immutable and thus easier to handle.

Hughes and Pareto [15] proposed a type and effect system on space usage estimation for a first-order functional language, extended with region language constructs of Tofte and Talpin's[20].The use of region model facilitates recovery of heap space. However, as each region is only deleted when all its objects become dead, more memory than necessary may be used, as reported by [4].

Hofmann and Jost [14] proposed a solution to obtain linear bounds on the heap space usage of first-order functional programs. A key feature of their solution is the use of linear typing which allows the space of each last-use data constructor (or record) to be directly recycled by a matching allocation. With this approach, memory recovery can be supported within each function, but *not across functions* in general. Moreover, their model does not track the symbolic sizes of data structures. Nevertheless, one significant advance of their work is an inference mechanism through linear programming (LP) technique.The main advantage of LP technique is that no fix-point analysis is required, but it restricts the memory effects to a linear form without disjunction.

Apart from the above memory analysis work on high level languages, Aspinall and Compagnoni [3] presented a first-order linearly typed assembly language to allow safe reuse of heap space.Their system is a target for the compilation of a functional programming language with a similar type systems (e.g. Hofmann's LFPL). More recently, Cachera et. al. [6] proposed a constraint-based memory analysis for Java Bytecode-like languages. For a given program their loop-detecting algorithm can detect methods and instructions that execute an unbounded number of times, thus can be used to check whether the memory usage is bounded or not. However, their analysis cannot check whether a given amount of memory is adequate or not, while our system does.

9 Concluding Remarks

We have proposed a memory usage type system for a non-trivial object-oriented core language. We have designed a flexible specification mechanism to allow memory needs of user programs to be declared abstractly, and then verifies if memory adequacy property holds for the given definitions. Our approach requires heap space to be explicitly deallocated, which can be handled automatically. We have also built a prototype type checker to confirm the viability and practicality of our approach. We envision our framework to be useful for embedded system, where memory is considered to be a critical resource. We also envision the synergy of predicable memory bounds with region-based

memory management systems. In particular, bounded memory regions can result in better performance. Synergistically, region-based system can provide timely recovery for shared objects that are dead, providing us with tighter memory bounds.

Acknowledgement. The authors would like to acknowledge the invaluable help of Florin Craciun with the evaluation of a set of the benchmark programs.

References

1. J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotation for Program Understanding. In *ACM OOPSLA*, Seattle, Washington, November 2002.
2. B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *ACM OOPSLA*, Denver, Colorado, November 1999.
3. D. Aspinall and A. Compagnoni. Heap bounded assembly language. *Journal of Automated Reasoning*, 31:261–302, 2003.
4. E. D. Berger, B. G. Zorn, and K. S. Mckinley. Reconsidering Custom Memory Allocation. In *ACM OOPSLA*, November 2002.
5. J. Boyland, J. Noble, and W. Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *ECOOP*, Budapest, Hungary, June 2001.
6. D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified Memory Usage Analysis. In *13th International Symposium of Formal Methods Europe (FM'05)*, July 2005.
7. M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *4th Principles and Practice of Parallel Programming*, Santa Barbara, California, May 1993.
8. E. C. Chan, J. Boyland, and W. L. Scherlis. Promises: Limited Specifications for Analysis and Manipulation. In *Proceedings of the International Conference on Software Engineering*, pages 167–176, Kyoto, Japan, April 1998.
9. W.N. Chin, S.C. Khoo, S.C. Qin, C. Popeea, and H.H. Nguyen. Verifying Safety Policies with Size Properties and Alias Controls. In *27th International Conference on Software Engineering (ICSE05)*, St. Louis, Missouri, May 2005.
10. W.N. Chin, H.H. Nguyen, S.C. Qin, and M. Rinard. Predictable Memory Usage for Object-Oriented Programs. Technical report, SoC, Natl Univ. of Singapore, November 2004. avail. at http://www.dur.ac.uk/shengchao.qin/papers/memj.ps.gz.
11. M. V. Christiansen and P. Velschow. Region-Based Memory Management in Java. Master's Thesis, Department of Computer Science (DIKU), University of Copenhagen, 1998.
12. M. Fahndrich and R. Leino. Declaring and checking non-null types in an object-oriented language. In *ACM OOPSLA*, Anaheim, CA, October 2003.
13. C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
14. M. Hofmann and S. Jost. Static prediction of heap space usage for first order functional programs. In *ACM POPL*, New Orleans, Louisiana, January 2003.
15. J. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space: Towards Embedded ML Programming. In *Proceedings of the International Conference on Functional Programming (ICFP '99)*, September 1999.
16. IBM. Jikes™ Research Virtual Machine (RVM). http://www-124.ibm.com/developerworks/oss/jikesrvm/.
17. L. Lamport. The temporal logic of actions. *ACM Trans. on Programming Languages and Systems*, 16(3):872–923, May 1994.
18. S Peyton-Jones and et al. Glasgow Haskell Compiler. http://www.haskell.org/ghc.

19. W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.
20. M. Tofte and J. Talpin. Region-based memory management. *Information and Computation*, 132(2), 1997.
21. H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *ACM PLDI*. ACM Press, June 1998.

A Alias Checking

We introduce four alias control mechanisms U | S | R | L adopted from [5, 8, 1]. These alias mechanisms shall be used to support precise size tracking in the presence of mutable objects, and also for the automatic recovery of dead unique objects. For size-tracking, we introduce R-mode fields to allow size-immutable properties to be accurately tracked for all objects. For example, an alternative class declaration for the list data type is given below, where its `next` field is marked as read-only (or immutable). Note that the `val` field remains mutable.

$$\texttt{class RList}\langle n\rangle \texttt{ where } n{=}m{+}1 \;;\; n{\geq}0 \;\{\; \texttt{Object}\langle\rangle\texttt{@S val};\; \texttt{RList}\langle m\rangle\texttt{@R next};\; \cdots \;\}$$

The size property of such an `RList` type can be analysed at compile-time, while allowing its objects to be freely shared. However, this comes at the cost of losing both mutability and uniqueness.

We make use of L-mode parameters, with the *limited unique* (or *lent-once*) property [8], to capture unique references that are temporarily lent out to method calls. They allow the preservation of uniqueness together with precise size-tracking across methods. Consider the following method with two `List` parameters.

$$\begin{array}{l}\texttt{void}\langle\rangle\texttt{@S join}(\texttt{List}\langle m\rangle\texttt{@L x},\ \texttt{List}\langle n\rangle\texttt{@U y})\ \texttt{where}\ n>0; m'{=}n{+}m; \cdots \\ \quad \{\ \texttt{if isNull}(\texttt{x.next})\ \texttt{then x.next} = \texttt{y else join}(\texttt{x.next}, \texttt{y})\ \}\end{array}$$

The first parameter is annotated as *lent-once* and can thus be tracked for size properties without loss of uniqueness. However, the second parameter is marked *unique* as its reference escapes the method body (into the tail of the `List` from the first parameter). In other words, the parameter `y` can have its uniqueness consumed but not `x`, as reflected in the body of the above method declaration. Given two unique lists, `a` and `b`, the call `join(a, b)` would consume the uniqueness of `b` but not that of `a`. Our lent-once policy is more restrictive than normal lending [1] as we require each lent-once parameter to be unaliased within the scope of its method. For example, `join(a, a)` is allowed by the type rules of [1], but disallowed by our lent-once's policy.

In our alias type system, uniqueness may be transferredfrom one location (variable, field or parameter) to another location. Consider a type environment $\{\texttt{x}::\texttt{Object}\langle\rangle\texttt{@U}, \texttt{y}::\texttt{Object}\langle\rangle\texttt{@U}, \texttt{z}::\texttt{Object}\langle\rangle\texttt{@S}\}$ where variables `x` and `y` are unique, while `z` is shared. In the following code, $\{\texttt{x} = \texttt{y}; \texttt{z} = \texttt{x}\}$, the uniqueness of `y` is first transferred to location `x`, followed by the consumption of uniqueness of `x` that is lost to the shared variable `z`. In our type judgement, we track variables/fields that have become dead using:

$$\Gamma; \Theta \vdash e :: t, \Theta_1$$

Here, each dead-set $\Theta(\Theta_1)$ captures the set of references with consumed uniqueness before(after) the evaluation of expression e. Γ is a type enviroment which maps variables to their annotated types. Other type judgements for methods, classes and programs have the following forms.

$$\Gamma \vdash_{meth} meth \qquad \vdash_{def} def \qquad \vdash_{P} def_{i:1..p}\ meth_{i:1..q}$$

The full set of alias checking rules are givenin our technical report [10]).

Abstraction Refinement for Termination

Byron Cook[1], Andreas Podelski[2], and Andrey Rybalchenko[2]

[1] Microsoft Research, Cambridge
[2] Max-Planck-Institut für Informatik, Saarbrücken

Abstract. Abstraction can often lead to spurious counterexamples. Counterexample-guided abstraction refinement is a method of strengthening abstractions based on the analysis of these spurious counterexamples. For invariance properties, a counterexample is a finite trace that violates the invariant; it is spurious if it is possible in the abstraction but not in the original system. When proving termination or other liveness properties of infinite-state systems, a useful notion of spurious counterexamples has remained an open problem. For this reason, no counterexample-guided abstraction refinement algorithm was known for termination. In this paper, we address this problem and present the first known automatic counterexample-guided abstraction refinement algorithm for termination proofs. We exploit recent results on transition invariants and transition predicate abstraction. We identify two reasons for spuriousness: abstractions that are too coarse, and candidate transition invariants that are too strong. Our counterexample-guided abstraction refinement algorithm successively weakens candidate transition invariants and refines the abstraction.

1 Introduction

The correctness argument for a program can sometimes be based on a small fraction of the original program code. However, it is often hard to extract this core automatically if the program is large and complex.

Automated abstraction refinement [6,19] is designed to solve precisely this problem. It automatically extracts just the information that is needed to prove the correctness property. Such algorithms are known for safety and invariance properties [2,5,6,13,14,15,16,17,19]. However, no such algorithm is known for termination proofs of infinite-state systems.

Abstraction refinement is based on the notion of spurious counterexamples. For invariance properties, a counterexample is a finite trace that violates the invariant. The counterexample is spurious if the trace is possible in the abstract system, but infeasible in the concrete system. The proof of the infeasibility of the trace provides guidance for adding more precision to the abstraction (and thus refining it).

For termination and liveness properties of infinite-state programs, a useful notion of spurious counterexamples has been an open problem. In this paper, we address this problem and present the first known counterexample-guided abstraction refinement algorithm for termination.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 87–101, 2005.
© Springer-Verlag Berlin Heidelberg 2005

We follow a recent approach to temporal verification of infinite-state systems that is based on transition invariants [21] and transition predicate abstraction [22]. This approach is a promising starting point for the development of our refinement method because of its connection with abstraction methods [11]. Let T be the transition relation of the infinite-state system. Transition invariant is the least fixed point of an operator $\mathcal{F}$ (defined as $\mathcal{F}(Q) = Q \circ T$), or rather its abstraction wrt. a set of transition predicates. The least fixed point construction is in analogy with abstract proofs for invariance properties, but the analogy stops here. Let us explain this point in detail.

Let I be an invariant and F be an operator such that

$$F(X) = \{s' \mid s \in X \text{ and } (s, s') \in T\} .$$

To prove that the invariant I holds we can search for an abstraction $F_P^{\#}$ based on a set of predicates P such that the least fixed point of $F_P^{\#}$ is contained in I:

$$\exists P. \quad \mathsf{lfp}(F_P^{\#}) \subseteq I .$$

The termination property does not come with an a priori fixed transition invariant. Any transition invariant is sufficient. Again, let $\mathcal{F}$ be $\mathcal{F}(Q) = Q \circ T$. In addition to finding an abstraction $\mathcal{F}_{\mathcal{P}}^{\#}$ of $\mathcal{F}$, we need to find a transition invariant $\mathcal{R}$ such that the least fixed point of $\mathcal{F}_{\mathcal{P}}^{\#}$ is contained in $\mathcal{R}$:

$$\exists \mathcal{R}\, \exists \mathcal{P}. \quad \mathsf{lfp}(\mathcal{F}_{\mathcal{P}}^{\#}) \subseteq \mathcal{R} . \tag{1}$$

The existence of the transition invariant $\mathcal{R}$ implies termination if $\mathcal{R}$ satisfies an additional property that we will explain later.

Thus, an automated termination checker that implements counterexample-guided abstraction refinement must not only construct an appropriate set of transition predicates $\mathcal{P}$, but also an assertion $\mathcal{R}$ that is an appropriate transition invariant. When the inclusion (1) does not hold, we do not know whether the left side is too big or the right side is not big enough. Thus, our refinement algorithm analyzes the reason why $\mathsf{lfp}(\mathcal{F}_{\mathcal{P}}^{\#})$ is not included in $\mathcal{R}$. Then, it chooses accordingly one of two possible actions. Either it decides that the abstraction is too coarse and it refines the abstraction by adding more transition predicates to $\mathcal{P}$ and thus makes $\mathsf{lfp}(\mathcal{F}_{\mathcal{P}}^{\#})$ smaller, or it decides that the candidate transition invariant $\mathcal{R}$ is too strong and weakens it.

This leads to a notion of counterexample that reflects both aspects of spuriousness: A counterexample is spurious if either the abstraction is too coarse or the candidate transition invariant is too strong. It is this new notion of spurious counterexamples that leads to the first known counterexample-guided abstraction refinement for the automation of termination proofs.

2 Related Work

Our work builds upon and benefits from the previous research on abstraction refinement (e.g. [2,5,6,13,14,15,16,17,19]) and automatic termination proofs

(e.g [4,8,10,12,20]) for infinite-state systems. A short way to distinguish our work from the existing literature in those two research areas is that we are the first to discover a method of abstraction refinement for termination analysis of infinite-state systems.

For the comparison with existing abstraction refinement tools: none of those tools can automatically prove termination, except for in trivial cases. This limitation is inherent to *predicate* abstraction (see [22] for an explanation).

The approach in [1] is to encode ranking functions into fairness assumptions for a finite model obtained by predicate abstraction; in contrast with our work, the actual termination arguments are ranking functions (which are found manually or by the above-mentioned tools without abstraction refinement).

The work in [22] presents an algorithm that, for a given set of *transition predicates*, constructs an abstraction of a program for the verification of liveness properties. This work does not, however, provide any guidance on how to refine the abstraction if it fails to prove the property.

Other proof methods for liveness properties have been proposed that are limited to only finite-state systems. For example the work in [3] exploits the fact that a non-terminating finite-state system must visit the same state infinitely many times.

3 Preliminaries

Programs. Following [18], we abstract away from the syntax of a concrete programming language such as C and represent a program P by a set of transitions. Each transition τ (to be thought of as the label of a program statement) refers to a transition constraint ρ_τ, which is an assertion over the program variables and their primed versions.

We use V and V' to represent the set of variables of the program and the set of their primed versions, respectively. The intended semantics of V' is to refer to the values of the variables V after executing a transition. The set V includes the variable pc (the program counter) which ranges over the program *locations*.

Each transition τ refers to a pair (ℓ, ℓ') of *pre* and *post* locations, respectively. These locations appear in the transition constraint ρ_τ in the form of the conjuncts $\mathsf{pc} = \ell$ and $\mathsf{pc}' = \ell'$. The program has an *initial location* ℓ^0 and an *initial condition* Θ, which is an assertion over program variables. The initial location ℓ^0 appears in Θ as the conjunct of the form $\mathsf{pc} = \ell^0$.

We assume that the program P is fixed from now on.

Program Semantics. A program *state* s is a valuation of the program variables, including the program counter pc.

We identify an *assertion over program variables* with the set of *states that it denotes*. For example, Θ is the the set of initial states. We also identify an *assertion over primed and unprimed program variables* with the *set of pairs of states that it denotes*. For example, ρ_τ is the transition relation of the transition τ.

A *computation* $s_0, s_1, s_2, \ldots$ is a possibly infinite sequence of states that starts in an initial state ($s_0 \in \Theta$) and that is consecutive, *i.e.*, each pair of successive

states belongs to the transition relation of some transition. Formally, for each $i \geq 0$ there exists a transition τ such that $(s_i, s_{i+1}) \in \rho_\tau$.

Paths and Cyclic Paths. A *path* $\pi = \tau_1 \ldots \tau_n$ is a (finite) sequence of transitions with consecutive locations (the post location of τ_i is the pre location of τ_{i+1}). A *cyclic path* $\pi = \tau_1 \ldots \tau_n$ is a special case of a path with the same start and end location (the pre location of its first transition τ_1 is equal to the post location of its last transition τ_n).[1]

We define the composition of relations $\rho_1 \circ \rho_2$ as usual:

$$\rho_1 \circ \rho_2 \quad \equiv \quad \{(s, s') \mid (s, s'') \in \rho_1 \text{ and } (s'', s') \in \rho_2\} \ .$$

It can be implemented by logical operations over transition constraints.

A path π denotes a transition relation ρ_π that is naturally obtained by composing the transition relations of the transitions along the path. Formally, for a path $\pi = \tau_1 \ldots \tau_n$ we have:

$$\rho_\pi \quad \equiv \quad \rho_{\tau_1} \circ \cdots \circ \rho_{\tau_n} \ .$$

Termination. A program is *terminating* if it does not admit any infinite computation. A binary relation R is *well-founded* if there exists no infinite sequence $s_0, s_1, s_2, \ldots$ that is consecutive wrt. R (formally, for each $i \geq 0$ we have $(s_i, s_{i+1}) \in R$).

The following fact is a consequence of Theorem 1 in [21] (by the fact that the transition relation of each path π with different start and end locations ℓ and ℓ' is contained in the well-founded relation $R_{\ell,\ell'} \ \equiv \ \mathsf{pc} = \ell \wedge \mathsf{pc}' = \ell'$).

Theorem 1 (Termination Condition [21]). *The program is terminating if there exists a finite set of well-founded relations $\mathcal{R} = \{R_1, \ldots, R_m\}$ such that the transition relation ρ_π of each cyclic path $\pi = \tau_1 \ldots \tau_n$ is included in one of the relations from $\mathcal{R}$.*

The termination condition in the theorem above is formally:

$$\text{for each cyclic path } \pi = \tau_1 \ldots \tau_n : \quad \rho_\pi \subseteq R_1 \text{ or } \ldots \text{ or } \rho_\pi \subseteq R_m \ . \qquad (2)$$

Transition Predicates. We use transition predicate abstraction [22] in order to obtain a termination condition that is stronger than (2), and that can be checked effectively. A *transition predicate* p is an assertion over program variables and their primed version, *i.e.*, p is a binary relation over states. In contrast, a (plain) predicate is an assertion over program variables, *i.e.*, a set of states. We use $\mathcal{P}$ to refer to a finite set of transition predicates. *Transition predicate abstraction* is similar to predicate abstraction if one replaces the set of program variables V by the set $V \cup V'$.

[1] Note that a cyclic path with end location ℓ may have numerous other steps that pass through ℓ.

An *abstraction function* $\alpha_{\mathcal{P}}$ maps a binary relation ρ over states to a superset expressed by a conjunction of transition predicates. We assume that one can automatically construct the abstraction function $\alpha_{\mathcal{P}}$ for a given finite set of transition predicates $\mathcal{P}$. A possible definition is the abstraction of a relation ρ by the conjunction of all transition predicates $p \in \mathcal{P}$ weaker than ρ (and test the 'weaker-than' relation $\rho \models p$ with a theorem prover).

For our formal treatment in Theorem 3, we will use one basic fact about the abstraction function $\alpha_{\mathcal{P}}$: the abstraction of a relation ρ is the relation itself (*i.e.* there is no loss of precision during abstraction) if ρ can be expressed by a conjunction of transition predicates (see Theorem 13 in [9]). Formally,

$$\alpha_{\mathcal{P}}(\rho) = \rho \quad \text{if } \rho = p_1 \wedge \ldots \wedge p_n \quad \text{for } p_1, \ldots, p_n \in \mathcal{P} \ . \tag{3}$$

Abstraction of Paths. We can construct an abstraction $\widehat{\alpha}_{\mathcal{P}}(\pi)$ for each path $\pi = \tau_1 \ldots \tau_n$ according to the following inductive definition.

$$\begin{aligned} \widehat{\alpha}_{\mathcal{P}}(\tau_1 \ldots \tau_n) &\equiv \alpha_{\mathcal{P}}(\rho_{\tau_1} \circ \rho) \quad \text{where } \rho = \widehat{\alpha}_{\mathcal{P}}(\tau_2 \ldots \tau_n) \\ \widehat{\alpha}_{\mathcal{P}}(\tau_n) &\equiv \alpha_{\mathcal{P}}(\rho_{\tau_n}) \end{aligned}$$

The abstraction of the path π is always a superset of the transition relation of π, formally

$$\rho_\pi \subseteq \widehat{\alpha}_{\mathcal{P}}(\pi) \ .$$

We obtain a termination condition that is effective in the sense that one can compute an abstraction $\widehat{\alpha}_{\mathcal{P}}(\pi)$ of each possible (cyclic) path π.

Theorem 2 (Termination Condition with Abstraction [22]). *The program is terminating if there exists a finite set of well-founded relations* $\mathcal{R} = \{R_1, \ldots, R_m\}$ *such that* the abstraction $\alpha_{\mathcal{P}}(\pi)$ *of the transition relation of each cyclic path* $\pi = \tau_1 \ldots \tau_n$ *is included in one of the relations from* $\mathcal{R}$*.*

This 'effective' termination condition is formally:

$$\text{for each cyclic path } \pi = \tau_1 \ldots \tau_n : \ \widehat{\alpha}_{\mathcal{P}}(\pi) \subseteq R_1 \text{ or } \ldots \text{ or } \widehat{\alpha}_{\mathcal{P}}(\pi) \subseteq R_m \ . \tag{4}$$

For notational convenience, we overload the symbol $\alpha_{\mathcal{P}}$. We will use $\alpha_{\mathcal{P}}$ not only as a function on relations ρ, but also as a function $\widehat{\alpha}_{\mathcal{P}}$ over paths π. We need to distinguish the two functions. The abstraction of the transition relation ρ_π is in general a subset of the abstraction of the path π, formally,

$$\alpha_{\mathcal{P}}(\rho_\pi) \subseteq \alpha_{\mathcal{P}}(\pi) \ .$$

For example, given the transition relations

$$\begin{aligned} \rho_{\tau_1} &\equiv x' = x - 2, \\ \rho_{\tau_2} &\equiv x' = x + 1, \end{aligned}$$

and a singleton set of transition predicates

$$\mathcal{P} = \{x' \leq x\} \ ,$$

we have

$$\begin{aligned}\alpha_{\mathcal{P}}(\rho_{\tau_1\tau_2}) &= \alpha_{\mathcal{P}}(\rho_{\tau_1} \circ \rho_{\tau_2}) \\ &= \alpha_{\mathcal{P}}(x' = x - 1) \\ &= x' \leq x \ ,\end{aligned}$$

whereas

$$\begin{aligned}\alpha_{\mathcal{P}}(\tau_1\tau_2) &= \alpha_{\mathcal{P}}(\rho_{\tau_1} \circ \alpha_{\mathcal{P}}(\tau_2)) \\ &= \alpha_{\mathcal{P}}(\rho_{\tau_1} \circ \mathsf{true}) \\ &= \alpha_{\mathcal{P}}(\mathsf{true}) \\ &= \mathsf{true} \ .\end{aligned}$$

Thus, we have $\alpha_{\mathcal{P}}(\rho_{\tau_1\tau_2}) \subsetneq \alpha_{\mathcal{P}}(\tau_1\tau_2)$.

4 Refinement for Termination

The termination condition (4) suggests that, given a set of well-founded relations $\mathcal{R} = \{R_1, \ldots, R_m\}$, the problem of refinement is to find the 'right' set of transition predicates $\mathcal{P}$. The set $\mathcal{P}$ is 'right' if the induced abstraction $\alpha_{\mathcal{P}}$ is sufficiently precise to infer an inclusion of the form $\alpha_{\mathcal{P}}(\pi) \subseteq R_j$ for every cyclic path π, see (4).

Our algorithm must, however, also find the 'right' set of well-founded relations $\mathcal{R} = \{R_1, \ldots, R_m\}$. The set $\mathcal{R}$ is 'right' if the inclusion $\rho_\pi \subseteq R_j$ holds 'in the concrete' for every path π, see (2).

Counterexamples. Distinction between the two cases above complicates the notion of a spurious counterexample. Namely, if the abstract check (4) does not succeed for a cyclic path π, then this may be spurious for one of two reasons: either the set of transition predicates $\mathcal{P}$ was not yet 'right' or the set of well-founded relations $\mathcal{R}$ was not yet 'right'.

Definition 1 (Spurious Counterexample). *Given a set of transition predicates $\mathcal{P}$ and a set of well-founded relations $\mathcal{R} = \{R_1, \ldots, R_m\}$, a cyclic path $\pi = \tau_1 \ldots \tau_n$ is a* counterexample *wrt. $\mathcal{P}$ and $\mathcal{R}$ if its abstraction $\alpha_{\mathcal{P}}(\pi)$ is not contained in any relation in $\mathcal{R}$. Formally,*

$$\alpha_{\mathcal{P}}(\pi) \not\subseteq R_j \quad \textit{for each } j \in \{1, \ldots, m\} \ .$$

The counterexample π is spurious *if either its relation ρ_π is contained in some relation R_j of $\mathcal{R}$ or its relation ρ_π is well-founded. Formally,*

$$\rho_\pi \subseteq R_j \quad \textit{for some } j \in \{1, \ldots, m\} \quad \textit{or} \quad \rho_\pi \ \textit{well-founded.}$$

The Algorithm. Figure 1 shows our counterexample-guided abstraction refinement for termination. For each new set of well-founded relations $\mathcal{R}$ and for each new set of transition predicates $\mathcal{P}$, the algorithm checks whether there exists a counterexample wrt. $\mathcal{R}$ and $\mathcal{P}$. It does so by going through all cyclic paths π until no more new abstract values $\alpha_{\mathcal{P}}(\pi)$ can be computed. Although the number of cyclic paths is infinite, the search converges because the range of the abstraction

```
1   input
2     Program P
3   begin
4     𝓡 := ∅                                  (* set of well-founded relations *)
5     𝓟 := ∅                                  (* set of transition predicates *)
6     repeat
7       if exists π = τ_1 ... τ_n s.t. α_𝓟(π) ⊈ R for any R ∈ 𝓡 then
8         if exists R ∈ 𝓡 such that ρ_π ⊆ R then
9           (* refinement step *)
10          𝓟_path := ⋃_{i∈1..n} Preds(ρ_{τ_i} ∘ ··· ∘ ρ_{τ_n})
11          𝓟_loop := Preds(R) ∪ ⋃_{i∈1..n} Preds(ρ_{τ_i} ∘ ··· ∘ ρ_{τ_n} ∘ R)
12          𝓟 := 𝓟 ∪ 𝓟_path ∪ 𝓟_loop
13        else
14          if π is well-founded by the ranking relation R then
15            (* weakening step *)
16            𝓡 := 𝓡 ∪ {R}
17          else
18            return "Counterexample cyclic path τ_1 ... τ_n"
19      else
20        return "Program P terminates"
21  end.
```

Fig. 1. Counterexample-guided abstraction refinement for termination. In line 7, we investigate abstractions $\alpha_{\mathcal{P}}(\pi)$ of cyclic paths by exploring the paths in a breadth-first way. The exploration converges since the range of the abstraction function $\alpha_{\mathcal{P}}$ is finite. In line 10, $\mathsf{Preds}(T)$ symbolically evaluates T and then extracts the set of atomic formulas from the reduced expression.

function $\alpha_{\mathcal{P}}$ is finite (and determined by the number of transition predicates in $\mathcal{P}$).

If the algorithm finds no counterexample, it has succeeded in proving the termination property and it stops. If the algorithm finds a counterexample π, there are three possibilities.

1. The counterexample π is spurious because the set of transition predicates was not yet 'right'. Formally, the inclusion between ρ_π and some $R \in \mathcal{R}$ does not hold in the abstract, *i.e.* $\alpha_{\mathcal{P}}(\pi) \not\subseteq R$, but it does hold in the concrete, *i.e.* $\rho_\pi \subseteq R$. The refinement step adds a set of transition predicates $\mathcal{P}_{\text{path}}$ from the transition relation of every suffix of the path $\pi = \tau_1 \dots \tau_n$ to the set $\mathcal{P}$. These predicates will eliminate this particular counterexample in the next iteration of the algorithm. The set of predicates $\mathcal{P}_{\text{loop}}$ guarantees that the refinement will not get 'stuck in a loop' discovering an infinite sequence of counterexamples π, $\pi\pi$, $\dots$, π^i, $\dots$. We will provide a formal statement describing the progress of refinement in Theorem 3.
2. The counterexample π is spurious because the set of well-founded relations $\mathcal{R}$ was not yet 'right'. This means that for any $R \in \mathcal{R}$ the inclusion $\rho_\pi \subseteq R$ does not hold neither in the abstract nor in the concrete, but the transi-

tion relation ρ_π of the cyclic path π is well-founded. This means that the candidate set $\mathcal{R}$ is not yet 'right'. In that case a well-founded relation R containing ρ_π is added to $\mathcal{R}$. In the next iteration of the algorithm, the same counterexample π may be found again, but then we will be in Case 1.

3. The counterexample π is not spurious: the transition relation ρ_π of the cyclic path π is not well-founded. In that case, the algorithm has failed to prove the termination property and it stops. In this case π^ω may be a feasible infinite trace.

Well-Foundedness and Ranking Relations. A ranking function for a (terminating) program is defined by an expression rank over the program variables. Its value for each reachable program state is a non-negative integer that decreases during each computation step.

We write $\mathsf{rank}(V)$ for the expression in the program variables and $\mathsf{rank}(V')$ for the expression in the primed version of the program variables. A ranking function defined by the expression rank induces a well-founded relation (a *ranking relation*) R in the following way:

$$R \equiv \mathsf{rank}(V) \geq 0 \wedge \mathsf{rank}(V') \leq \mathsf{rank}(V) - 1 .$$

We note the following observation.

Remark 1. A ranking relation R is transitive. Formally,

$$R \circ R \subseteq R .$$

A cyclic path $\pi = \tau_1 \dots \tau_n$ defines a program fragment of a very specific form: it consists of one program location ℓ and one transition from ℓ to ℓ with the transition relation ρ_π. There exist several automatic methods and tools for the computation of ranking functions for such programs, *e.g.* [4,7,20,24]. These tools can be used for implementing line 14 of the algorithm.

Progress of Refinement. A newly detected *spurious* counterexample gives rise to a new refinement step and a new iteration of the algorithm. The refinement algorithm makes progress if for each newly detected *spurious* counterexample π the cyclic path π is no longer a counterexample after the next iteration or the next two iterations of the algorithm. Our algorithm enjoys the property of eliminating the infinite set of spurious counterexamples $\pi, \pi\pi, \dots$ in a single step. We formalize this property in Theorem 3.

Theorem 3 (Progress of Refinement). *If π is a spurious counterexample wrt. the sets $\mathcal{R}$ and $\mathcal{P}$, then none of the cyclic paths $\pi_1, \pi_2, \dots$ obtained by concatenating π with itself repeatedly ($\pi_1 = \pi$, $\pi_2 = \pi\pi$, etc.) is a counterexample wrt. the sets $\mathcal{R}'$ and $\mathcal{P}'$ obtained by refinement in one or possibly two more iterations of the algorithm in Figure 1.*

Proof. Given a spurious counterexample $\pi = \tau_1 \dots \tau_n$, there are two cases that we need to consider. In the first case, the relation ρ_π is included in some $R \in \mathcal{R}$

(at line 8 on Figure 1). Hence, the refinement step (at lines 10, 11, and 12) updates the abstraction function. Now we consider the next iteration of the algorithm. Let $\mathcal{P}'$ be the current set of transition predicates, which define the abstraction function.

We prove that $\alpha_{\mathcal{P}'}(\pi^j) \subseteq R$ by induction over j.[2] For the base case $j = 1$, we prove $\alpha_{\mathcal{P}'}(\pi) \subseteq R$. By Theorem 13 in [9], an abstraction function is precise for some input if the input is expressible by the predicates defining the abstraction. Hence, for each $i \in \{1, \dots, n\}$ we have $\alpha_{\mathcal{P}'}(\tau_i \dots \tau_n) = \rho_{\tau_i} \circ \cdots \circ \rho_{\tau_n}$. Thus, we have $\alpha_{\mathcal{P}'}(\pi) \subseteq R$.

For the induction step, we assume $\alpha_{\mathcal{P}'}(\pi^j) \subseteq R$ for some $j > 1$. By Theorem 13 in [9], we have $\alpha_{\mathcal{P}'}(\tau_i \dots \tau_n \pi^j) \subseteq \rho_{\tau_i} \circ \cdots \circ \rho_{\tau_n} \circ R$ for each $i \in \{1, \dots, n\}$. Hence, we have $\alpha_{\mathcal{P}}(\pi\pi^j) = \rho_\pi \circ R$. Since $\rho_\pi \subseteq R$ and by the assumption that R is a transitive relation, we have $\alpha_{\mathcal{P}}(\pi^{j+1}) \subseteq R \circ R \subseteq R$.

If ρ_π is not contained in any $R \in \mathcal{R}$, then after the weakening step at line 16 using a ranking relation R we have $\rho_\pi \subseteq R$, and the above case applies. □

5 Example

In this section we execute the algorithm contained in Figure 1 on a sample program fragment. Refer to left-hand side of Figure 2 for the example program. We represent the program as a control-flow graph on the right-hand side, where each node is the start of a basic-block, and each transition (labeled τ_1, τ_1, and τ_3) is decorated with a relation that represents the conditions and assignments of the basic block. We have the following transition relations ρ_{τ_i}:

$$\begin{aligned} \rho_{\tau_1} &\equiv x \geq 0 \land x' = x + 1 \land y' = 1 \land \mathsf{pc} = \ell_0 \land \mathsf{pc}' = \ell_1 \ , \\ \rho_{\tau_2} &\equiv y > x \land x' = x - 2 \land y' = y \land \mathsf{pc} = \ell_1 \land \mathsf{pc}' = \ell_0 \ , \\ \rho_{\tau_3} &\equiv y \leq x \land x' = x \land y' = y + 1 \land \mathsf{pc} = \ell_1 \land \mathsf{pc}' = \ell_1 \ . \end{aligned}$$

To simplify the presentation, we assume an implicit treatment of the program counter. This means that we do not show any predicates involving pc in the exposition below.

We summarize the intermediate steps of our example execution in Table 1, and give a detailed explanation below. Line numbers refer to the algorithm shown on Figure 1.

Step I/Line 4 and 5: We start with the empty set of well-founded relations $\mathcal{R} = \emptyset$ and the empty set of transition predicates $\mathcal{P} = \emptyset$.

Step II/Lines 7, 8, 10, 11, and 12: We start enumerating cyclic paths and computing their abstractions. Because $\mathcal{R}$ is empty, we find that for the cyclic path $\pi = \tau_1\tau_2$ the abstract relation $\alpha_{\mathcal{P}}(\pi)$ does not entail any relations in $\mathcal{R}$. This means that π is a counterexample. We do not know yet whether it is spurious. We therefore move to line 8. For the same reason there does not

[2] Note that we abstract wrt. a refined set of transition predicates $\mathcal{P}'$.

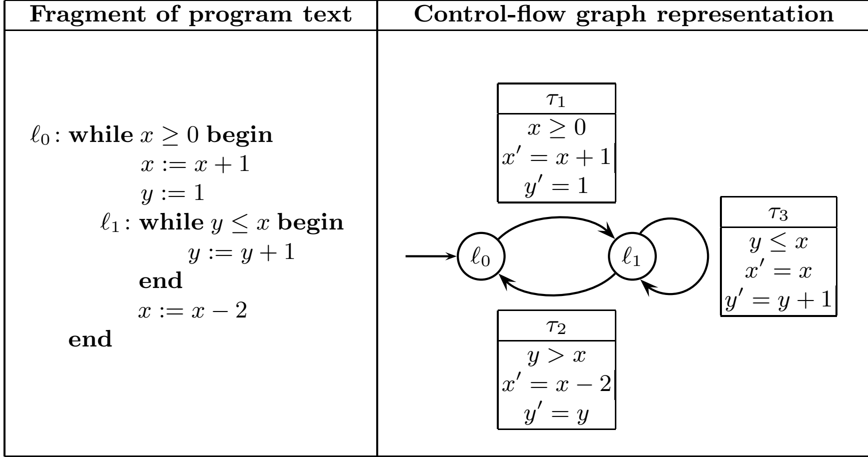


Fig. 2. Example program fragment with nested loops

Table 1. The states of the algorithm in Figure 1 while analyzing the example in Figure 2

Step	Path π	$\forall R \in \mathcal{R}$	Action
I	-	-	Initialization with $\mathcal{R} = \emptyset$ and $\mathcal{P} = \emptyset$
II	$\tau_1\tau_2$	$\rho_\pi \not\subseteq R$	Weakening with $R_1 = \mathsf{false}$
III	$\tau_1\tau_2$	$\alpha_{\mathcal{P}}(\pi) \not\subseteq R$	Refinement by $\mathsf{Preds}(\rho_{\tau_2}) = \{y > x, x' = x - 2, y' = y\}$, $\mathsf{Preds}(\rho_{\tau_1} \circ \rho_{\tau_2}) = \emptyset$, $\mathsf{Preds}(\dots R_1) = \emptyset$.
IV	τ_3	$\rho_\pi \not\subseteq R$	Weakening with $R_2 = x - y \geq 0 \wedge x' - y' \leq x - y - 1$
V	τ_3	$\alpha_{\mathcal{P}}(\pi) \not\subseteq R$	Refinement by $\mathsf{Preds}(\rho_{\tau_3}) = \{y \leq x, x' = x, y' = y + 1\}$, $\mathsf{Preds}(R_2) = \{x - y \geq 0, x' - y' \leq x - y - 1\}$, $\mathsf{Preds}(\rho_{\tau_3} \circ R_2) = \{y \leq x - 1, x' - y' \leq x - y - 2\}$.
VI	$\tau_2\tau_1$	$\rho_\pi \not\subseteq R$	Weakening with $R_3 = x \geq 2 \wedge x' \leq x - 1$
VII	$\tau_2\tau_1$	$\alpha_{\mathcal{P}}(\pi) \not\subseteq R$	Refinement by $\mathsf{Preds}(\rho_{\tau_1}) = \{x \geq 0, x' = x + 1, y' = 1\}$, $\mathsf{Preds}(\rho_{\tau_2} \circ \rho_{\tau_1}) = \{y > x, x \geq 2, x' = x - 1, y' = 1\}$, $\mathsf{Preds}(R_2) = \{x \geq 2, x' \leq x - 1\}$, $\mathsf{Preds}(\rho_{\tau_1} \circ R_2) = \{x \geq 1, x' \leq x\}$, $\mathsf{Preds}(\rho_{\tau_2} \circ \rho_{\tau_1} \circ R_2) = \{y > x, x \geq 3, x' \leq x - 3\}$.
VIII	$\tau_1\tau_3\tau_2$	$\rho_\pi \not\subseteq R$	Weakening with $R_4 = x \geq 0 \wedge x' \leq x - 1$

exist a relation R in $\mathcal{R}$ such that $\rho_\pi \subseteq R$. We therefore move to line 14. The composition $\rho_{\tau_1} \circ \rho_{\tau_2}$ equals

$$\begin{aligned}\rho_{\tau_1} \circ \rho_{\tau_2} &= \exists x''.\ x \geq 0 \wedge x'' = x + 1 \wedge y'' = 1 \wedge \\ &\qquad y'' > x'' \wedge x' = x'' - 2 \wedge y' = y'' \\ &= x \geq 0 \wedge 1 > x + 1 \wedge x' = x - 1 \wedge y' = 1 \\ &= x \geq 0 \wedge 1 > x + 1 \\ &= x \geq 0 \wedge 0 > x \\ &= \mathsf{false}\ .\end{aligned}$$

Since false is well-founded, the counterexample π is spurious because the candidate set $\mathcal{R}$ is too strong. The ranking relation that provides the evidence of ρ_π's well-foundedness is the empty relation. Hence, we go to line 16, and add the empty relation $R_1 \equiv \emptyset$ to $\mathcal{R}$.

Step III/Lines 7, 8, 10, 11, and 12: We observe that the cyclic path $\pi = \tau_1\tau_2$ is still a spurious counterexample, since

$$\begin{aligned}\alpha_{\mathcal{P}}(\pi) &= \alpha_{\mathcal{P}}(\rho_{\tau_1} \circ \alpha_{\mathcal{P}}(\tau_2)) \\ &= \alpha_{\mathcal{P}}(\rho_{\tau_1} \circ \alpha_{\mathcal{P}}(y > x \wedge x' = x - 2 \wedge y' = y)) \\ &= \alpha_{\mathcal{P}}(\rho_{\tau_1} \circ \mathsf{true}) \\ &= \alpha_{\mathcal{P}}(\mathsf{true}) \\ &= \mathsf{true}\ ,\end{aligned}$$

and because true does not entail R_1. We go to line 8. Recall that $\rho_{\tau_1} \circ \rho_{\tau_2} = \mathsf{false}$. Because $\mathsf{false} \subseteq R_1$, we detect that the counterexample π is spurious due to imprecise abstraction. Hence, we go to line 10, and we collect the sets of predicates $\mathsf{Preds}(\rho_{\tau_2})$ and $\mathsf{Preds}(\rho_{\tau_1} \circ \rho_{\tau_2})$, see Table 1. The later set is empty, since $\rho_{\tau_1} \circ \rho_{\tau_2} = \mathsf{false}$. The set of predicates collected at line 11 is empty because R_1 is empty. Therefore, we finish this step with the following set of transition predicates:

$$\mathcal{P} = \{y > x, x' = x - 2, y' = y\}\ .$$

Step IV/Lines 7, 8, 14, and 16: We note that $\tau_1\tau_2$ is no longer a counterexample, because $\alpha_{\mathcal{P}}(\tau_1\tau_2) \subseteq R_1$. We find that for the cyclic path $\pi = \tau_3$ the abstract relation $\alpha_{\mathcal{P}}(\pi)$ does not entail any relations in $\mathcal{R}$. This means that π is a counterexample. We do not know yet whether it is spurious. We therefore move to line 8. There does not exist a relation R in $\mathcal{R}$ such that $\rho_\pi \subseteq R$. We therefore move to line 14. Recall that $\rho_{\tau_3} \equiv y \leq x \wedge x' = x \wedge y' = y + 1$. Using the techniques described in [20], we prove that ρ_{τ_3} is well-founded. We also compute a witness of ρ_{τ_3}'s well-foundedness. The witness is a ranking relation R_2 such that $\rho_{\tau_3} \subseteq R_2$. We have

$$R_2 \equiv x - y \geq 0 \wedge x' - y' \leq x - y - 1\ .$$

Hence, π is a spurious counterexample. We weaken $\mathcal{R}$ by adding R_2, at line 16.

Step V/Lines 7, 8, 10, 11, and 12: For $\pi = \tau_3$ we have $\alpha_{\mathcal{P}}(\pi) = \mathsf{true}$. Therefore $\alpha_{\mathcal{P}}(\pi)$ does not entail R_2. We know that $\rho_\pi \subseteq R_2$ (see Step IV). This means that τ_3 is still a (spurious) counterexample wrt. the current abstraction. We refine the abstraction. The condition at line 8 succeeds, and we move to line 10. We collect the predicates from $\mathsf{Preds}(\rho_{\tau_3})$. At line 11, we collect the predicates from $\mathsf{Preds}(R_2)$, and $\mathsf{Preds}(\rho_{\tau_3} \circ R_2)$. After executing line 11, we have

$$\mathcal{P} = \{y > x, x' = x - 2, y' = y, y \leq x, x' = x, y' = y + 1,\\ x' - y' \leq x - y - 1, y \leq x - 1, x' - y' \leq x - y - 2\} \ .$$

Step VI/Lines 7, 8, 14, and 16: We observe that τ_3 is no longer a counterexample, since $\alpha_{\mathcal{P}}(\tau_3) \subseteq R_2$. We consider the abstraction of the cyclic path $\pi = \tau_2\tau_1$. We have that $\alpha_{\mathcal{P}}(\pi)$ does not entail neither R_1 nor R_2. The relation ρ_π such that

$$\rho_\pi \ = \ y > x \wedge x \geq 2 \wedge x' = x - 1 \wedge y' = 1$$

is well-founded, but is not contained in any $R \in \mathcal{R}$. Hence, π is a spurious counterexample. Therefore we execute lines 14, and 16 of the algorithm, which weaken $\mathcal{R}$. A witness to the well-foundedness of ρ_π is a ranking relation R_3 such that

$$R_3 \ \equiv \ x \geq 2 \wedge x' \leq x - 1 \ .$$

After executing line 16, we have $\mathcal{R} = \{R_1, R_2, R_3\}$.

Step VII/Lines 7, 8, 10, 11, and 12: Although $\rho_{\tau_2} \circ \rho_{\tau_1} \subseteq R_3$ we have $\alpha_{\mathcal{P}}(\tau_2 \circ \tau_1) \not\subseteq R_3$. This means that the abstraction is too coarse. Therefore, we execute lines 10, 11, and 12. At line 10, we collect the sets of predicates $\mathsf{Preds}(\rho_{\tau_1})$ and $\mathsf{Preds}(\rho_{\tau_2} \circ \rho_{\tau_1})$. At line 11, we collect the sets $\mathsf{Preds}(R_3)$, $\mathsf{Preds}(\rho_{\tau_1} \circ R_3)$, and $\mathsf{Preds}(\rho_{\tau_2} \circ \rho_{\tau_1} \circ R_3)$.

Step VIII/Lines 7, 8, 14, and 16: We observe that $\tau_2\tau_1$ is no longer a (spurious) counterexample. We discover that the relation ρ_π corresponding to the cyclic path $\pi = \tau_1\tau_3\tau_2$ is well-founded, but is not contained in any relation $R \in \mathcal{R}$:

$$\rho_{\tau_1} \circ \rho_{\tau_3} \circ \rho_{\tau_2} \ = \ x = 0 \wedge x' = x - 1 \wedge y' = 2 \ .$$

This means that we found another spurious counterexample. Therefore we execute lines 8, 14 and 16. The ranking relation R_4 such that

$$R_4 \ \equiv \ x \geq 0 \wedge x' \leq x - 1$$

is a witness to the well-foundedness of $\rho_{\tau_1} \circ \rho_{\tau_3} \circ \rho_{\tau_2}$. After executing line 14, we have $\mathcal{R} = \{R_1, R_2, R_3, R_4\}$.

Final Result: For the abstraction $\alpha_{\mathcal{P}}(\pi)$ of every cyclic path π there exists a relation R in $\mathcal{R} = \{R_1, R_2, R_3, R_4\}$ such that $\alpha_{\mathcal{P}}(\pi)$ entails R. Therefore, the

algorithm terminates with $\mathcal{R} = \{R_1, R_2, R_3, R_4\}$ and the set of predicates $\mathcal{P}$ where

$$
\begin{array}{rcl}
R_1 & = & \mathsf{false}\ , \\
R_2 & = & x - y \geq 0 \wedge x' - y' \leq x - y - 1\ , \\
R_3 & = & x \geq 2 \wedge x' \leq x - 1\ , \\
R_4 & = & x \geq 0 \wedge x' \leq x - 1\ ,
\end{array}
$$

and

$$
\begin{array}{rl}
\mathcal{P} = \{ & x \geq 0, x \geq 1, x \geq 2, x \geq 3, \\
& y \leq x, y \leq x - 1, y > x, \\
& x' = x + 1, x' = x, x' = x - 1, x' = x - 2, \\
& x' \leq x, x' \leq x - 1, x' \leq x - 3 \\
& x' - y' \leq x - y - 1, x' - y' \leq x - y - 2, \\
& y' = y + 1, y' = y, y' = 1\}\ .
\end{array}
$$

6 Conclusion

Counterexample-guided abstraction refinement allows us to automatically extract just the information that is needed to prove the property. The crux of our abstraction refinement procedure for termination is the notion of a counterexample, and the different possible root causes when counterexamples are spurious.

We presented the first known counterexample-guided abstraction refinement algorithm for the proof of termination. We exploit recent results on transition invariants and transition predicate abstraction. Our counterexample-guided abstraction refinement algorithm successively weakens candidate transition invariants and successively refines abstractions.

Future work. We are working on an implementation of this algorithm in SLAM. Possible extensions of the algorithm presented here concern a wider class of properties (liveness with fairness assumptions) and a wider class of programs (concurrent and recursive programs); here the techniques described in [22] and in [23] can be useful.

Acknowledgment. We thank Tom Ball, Aaron Bradley, and Lenore Zuck for discussions.

References

1. I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *VMCAI'2005: Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*, pages 164–180. Springer, 2005.
2. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *IFM'2004: Fourth International Conference on Integrated Formal Methods*, volume 2999 of *LNCS*, pages 1–20. Springer, 2004.

3. A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *FMICS'02: Formal Methods for Industrial Critical Systems*, volume 66(2) of *ENTCS*, 2002.
4. A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI'2005: Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*. Springer, 2005.
5. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE'2003: Int. Conf. on Software Engineering*, pages 385–395, 2003.
6. E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, December 1999.
7. M. Colón and H. Sipma. Synthesis of linear ranking functions. In *TACAS'2001: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 67–81. Springer, 2001.
8. M. Colón and H. Sipma. Practical methods for proving program termination. In *CAV'2002: Computer Aided Verification*, volume 2404 of *LNCS*, pages 442–454. Springer, 2002.
9. P. Cousot. Partial completeness of abstract fixpoint checking. In *SARA'2000: Abstraction, Reformulation, and Approximation*, volume 1864 of *LNCS*, pages 1–15. Springer, 2000.
10. P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI'2005: Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*, pages 1–24. Springer, 2005.
11. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'1977: Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
12. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL'1978: Principles of Programming Languages*, pages 84–97. ACM Press, 1978.
13. S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *LICS'2001: Logic in Computer Science*, pages 51–60. IEEE, 2001.
14. J. Hatcliff and M. B. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In *CONCUR'2001: Concurrency Theory*, volume 2154 of *LNCS*, pages 39–58. Springer, 2001.
15. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL'2004: Principles of Programming Languages*, pages 232–244. ACM Press, 2004.
16. F. Ivancic, H. Jain, A. Gupta, and M. K. Ganai. Localization and register sharing for predicate abstraction. In *TACAS'2005: Tools and Algorithms for Construction and Analysis of Systems*, LNCS. Springer, 2005. To appear.
17. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *TACAS'2001: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 98–112. Springer, 2001.
18. Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety*. Springer, 1995.
19. K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV'2000: Computer Aided Verification*, volume 1855 of *LNCS*, pages 139–153. Springer, 2000.

20. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI'2004: Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.
21. A. Podelski and A. Rybalchenko. Transition invariants. In *LICS'2004: Logic in Computer Science*, pages 32–41. IEEE, 2004.
22. A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL'2005: Principles of Programming Languages*, pages 132–144. ACM Press, 2005.
23. A. Podelski, I. Schaefer, and S. Wagner. Summaries for while programs with recursion. In S. Sagiv, editor, *ESOP'2005: European Symposium on Programming*, volume 3444 of *LNCS*, pages 94–107. Springer, 2005.
24. A. Tiwari. Termination of linear programs. In *CAV'2004: Computer Aided Verification*, volume 3114 of *LNCS*, pages 70–82. Springer, 2004.

Data-Abstraction Refinement: A Game Semantic Approach*

Aleksandar Dimovski[1], Dan R. Ghica[2], and Ranko Lazić[1]

[1] Department of Computer Science, Univ. of Warwick, Coventry, CV4 7AL, UK
[2] School of Computer Science, Univ. of Birmingham, Birmingham, B15 2TT, UK

Abstract. This paper presents a semantic framework for data abstraction and refinement for verifying safety properties of open programs. The presentation is focused on an Algol-like programming language that incorporates data abstraction in its syntax. The fully abstract game semantics of the language is used for model-checking safety properties, and an interaction-sequence-based semantics is used for interpreting potentially spurious counterexamples and computing refined abstractions for the next iteration.

1 Introduction

Abstraction refinement has proved to be one of the most effective methods of automated verification of systems with very large state spaces, especially software systems. Current state-of-the art tools implementing abstraction refinement algorithms [5, 16] combine model checking and theorem proving: model checking is used to verify whether an *abstracted system* satisfies a property, while theorem proving is used to *refine the abstraction* using the counterexamples discovered by model checking. Since abstractions are *conservative over-approximations* the safety of any abstracted program implies the safety of the concrete program. The converse is not true, and the refinement process may not terminate if the concrete program has an infinite state space.

This paper introduces a purely semantic approach to (data) abstraction refinement, based on game semantics [2, 17]. In order to keep the presentation focussed, the main vehicle of our development is the language *Abstracted Idealized Algol* (AIA), an expressive programming language combining imperative features, locally-scoped variables and (call-by-name) procedures. The key feature of this language is the use of abstraction schemes at the level of data-types, which allows the writing of abstracted programs in a syntax similar to that of concrete programs. In fact, a concrete program is a particular abstracted program, in which all the abstractions are identities.

The following is a simple example illustrating this method. Consider the (concrete) program fragment below, which uses local variable x, *non-local* function f,

* This research is supported by the EPSRC (GR/S52759/01). The 3rd author is also supported by a grant from the Intel Corporation, and affiliated to the Mathematical Institute,Serbian Academy of Sciences and Arts, Belgrade.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 102–117, 2005.
© Springer-Verlag Berlin Heidelberg 2005

and a command abort which causes abnormal termination. Is this program safe for all instantiations of f, or is it possible for its execution to terminate abnormally? [1]

$$\mathsf{newint}\, x := 0 \,\mathsf{in}\, f(x := !x + 1, \mathsf{if}\, !x = 5 \,\mathsf{then\ abort\ else\ skip})$$

The procedure-call mechanism is by-name, so every call to the first argument increments x, and any call to the second uses the new value of x. So the program is not safe if function f uses its first argument precisely 5 times, then its second argument.

We approximate the set of integers by a finite set of partitioning intervals. Let the initial abstraction have only one partition. We denote this in the program by annotation (see Table 1):

$$\mathsf{newint}_{[]}\, x := 0 \,\mathsf{in}\, f(x := !x +_{[]} 1, \mathsf{if}\, !x = 5 \,\mathsf{then\ abort\ else\ skip})$$

A counterexample execution trace exists, corresponding to the function evaluating its second argument. During the execution of this argument, the value of x is not 0 but, because of the abstraction, possibly any integer, chosen nondeterministically. If the chosen value is 5 then abort occurs. Of course, this counterexample is spurious because it is made possible only by the nondeterminism caused by over-abstraction. However, the counterexample informs the refinement procedure that the abstraction of x needs to be improved. Iterations like this one are performed until we obtain

$$\mathsf{newint}_{[0,5]}\, x := 0 \,\mathsf{in}\, f(x := !x +_{[0,5]} 1, \mathsf{if}\, !x = 5 \,\mathsf{then\ abort\ else\ skip})$$

at which point a genuine counterexample is discovered, corresponding to the behaviour resulting in abnormal termination.

In addition to giving a precise account of data abstraction-refinement, this approach has several advantages compared to alternative approaches:

Modularity. To our knowledge, examples such as the one described before, cannot be generally handled by known inter-procedural abstraction-refinement techniques. [8] has cogently advocated a need for such techniques, and we believe that we are meeting the challenge, although our approach is technically different.

Completeness and correctness. A concrete representation of a fully abstract semantic model is guaranteed to be accurate and is set on a firm theoretical foundation.

Compositionality. The semantic model is denotational, i.e. defined recursively on the syntax, therefore the model of a larger program is constructed from the models of its constituting sub-programs. This entails an ability to model program fragments, containing non-locally defined procedures as in the example above. This feature is the key to *scalability*, the modeling and verification of software systems that are too large to be dealt with as a whole.

[1] $!x$ denotes dereferencing.

Efficiency. As already emphasised in previous work on games-based model checking [1], finite-state representations of strategies give models of programs often several orders of magnitude smaller than state-exploration based models, essentially due to the fact that the details of local-state manipulation are hidden during composition.

2 Abstracted Idealized Algol

The data types of AIA are booleans and *abstracted integers* ($\tau ::= \mathsf{bool} \mid \mathsf{int}_\pi$). We use π to denote computable binary predicates on $\mathbb{Z}$. The *abstractions* π range over computable equivalence relations (i.e. partitions) of the integers $\mathbb{Z}$. To say that $m, n \in \mathbb{Z}$ are in the same class of π, we write $m \approx_\pi n$.

The phrase types of AIA are base types of expressions, variables and commands ($\sigma ::= \mathsf{exp}\tau \mid \mathsf{var}\tau \mid \mathsf{com}$) and function types ($\theta ::= \sigma \mid \theta \rightarrow \theta$).

We say that a type is *concrete* if it contains no abstractions other than the identity abstraction $\kappa = \big\{\{i\} \mid i \in \mathbb{Z}\big\}$. For any type θ, we write $\widetilde{\theta}$ for the concrete type obtained by replacing all abstractions with κ. For simplicity, we write int_κ as simply int.

The syntax of the language consists of imperative features (local variables, assignment, dereferencing, sequencing, branching, iteration, skip and abort) and functional features (abstraction, application, recursion, arithmetic-logic constants and operators). It is convenient to present the syntax of AIA in a "functionalised" form [3], using function-constants rather than term combinators, as in:

$$\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, N \cong \mathsf{if}\, B\, M\, N$$

$$\mathsf{new}\tau\; x := E\; \mathsf{in}\; C \cong \mathsf{new}\, E\, (\lambda\, x : \mathsf{var}\tau . C),\ \textit{etc.}$$

Combinators can be reintroduced as syntactic sugar, to improve readability.

The base-type constants are:

$$\mathsf{true}, \mathsf{false} : \mathsf{expbool} \quad \mathsf{abort}, \mathsf{skip} : \mathsf{com} \quad n : \mathsf{expint}_\pi$$

The functional constants are:

$$\mathsf{new} : \mathsf{exp}\tau_1 \rightarrow (\mathsf{var}\tau_2 \rightarrow \mathsf{com}) \rightarrow \mathsf{com},\ \widetilde{\tau_1} = \widetilde{\tau_2} \qquad \mathsf{if} : \mathsf{expbool} \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \sigma,\ \widetilde{\sigma_1} = \widetilde{\sigma_2} = \widetilde{\sigma}$$

$$\mathsf{asg} : \mathsf{var}\tau_1 \rightarrow \mathsf{exp}\tau_2 \rightarrow \mathsf{com},\ \widetilde{\tau_1} = \widetilde{\tau_2} \qquad \mathsf{while} : \mathsf{expbool} \rightarrow \mathsf{com} \rightarrow \mathsf{com}$$

$$\mathsf{der} : \mathsf{var}\tau \rightarrow \mathsf{exp}\tau \qquad \mathsf{rec} : (\theta \rightarrow \theta) \rightarrow \theta$$

$$\mathsf{seq} : \mathsf{com} \rightarrow \sigma \rightarrow \sigma \qquad \mathsf{op} : \mathsf{exp}\tau_1 \rightarrow \mathsf{exp}\tau_2 \rightarrow \mathsf{exp}\tau$$

where op stands for any arithmetic-logic operator whose concrete type is $\mathsf{exp}\widetilde{\tau_1} \rightarrow \mathsf{exp}\widetilde{\tau_2} \rightarrow \mathsf{exp}\widetilde{\tau}$. For example, for any abstractions π_1 and π_2, AIA contains an equality operator $=$ of type $\mathsf{expint}_{\pi_1} \rightarrow \mathsf{expint}_{\pi_2} \rightarrow \mathsf{expbool}$.

For types of new, asg and if to be valid, it is required above that corresponding subterms of types have equal concretisations, but their abstractions can be

Notation for sets of integers:

$$<n = \{n' \mid n' < n\}, \quad n = \{n\}, \quad >n = \{n' \mid n' > n\}$$

Notation for abstractions:

$$[] = \{\mathbb{Z}\}, \qquad [n, m] = \{<n, n, n+1, \ldots, m-1, m, >m\}$$

Fig. 1. Some integer abstractions

different. For example, for any abstractions π_1 and π_2, we can assign expressions of type int_{π_2} to variables of type int_{π_1}. This flexibility, which is also present in the rule for functional application below, enables abstractions within a term to be changed independently of each other while preserving well-typed-ness.

$\Gamma \vdash M : \theta$, where Γ is a list of typed identifiers, indicates that term M with free identifiers in Γ has type θ. The typing rules are:

$$\Gamma \vdash \mathsf{k} : \theta \text{ (}\mathsf{k} \text{ is a language constant of type } \theta) \qquad \Gamma, x : \theta \vdash x : \theta$$

$$\frac{\Gamma, x : \theta \vdash M : \theta'}{\Gamma \vdash \lambda\, x : \theta . M : \theta \to \theta'} \quad \frac{\Gamma \vdash M : \theta_1 \to \theta \qquad \Gamma \vdash N : \theta_2}{\Gamma \vdash M\, N : \theta}\ \widetilde{\theta_1} = \widetilde{\theta_2}$$

Whenever we write $\Gamma \vdash M : \theta$, we are considering implicitly a particular derivation of that typing judgment from the rules above. Such a derivation contains typing judgments for all sub-terms of M. When we need to be explicit about which derivation was used, we shall annotate M with abstractions. For example, with the notations in Fig. 1,

$$x : \mathsf{varint}_{[0,4]} \vdash x := !x +_{[0,4] \to [0,1] \to [0,3]} 1_{[0,1]} : \mathsf{com}$$

means that the operator $+$ was used with type $\mathsf{expint}_{[0,4]} \to \mathsf{expint}_{[0,1]} \to \mathsf{expint}_{[0,3]}$. Here the combinators $:=$ and $!$ are syntactic sugar for applications of the functional constants asg and der.

We say that a term is *concrete* if it contains no abstractions other than the identity abstraction κ. For any term $\Gamma \vdash M : \theta$, we write $\widetilde{\Gamma} \vdash \widetilde{M} : \widetilde{\theta}$ for the concrete term obtained by replacing all abstractions with κ.

The operational semantics is defined as a big-step reduction relation $M, s \Longrightarrow \mathcal{K}$, where M is a *program* (all free identifiers are assignable variables), s is a *state* (a function assigning data values to the variables), and $\mathcal{K}$ is a final configuration. The final configuration can be either a pair V, s' with V a *value* (i.e. a language constant or an abstraction $\lambda\, x : \theta . M$) and s' a state, or special error configuration $\mathcal{E}$.

The reduction rules are similar to those for IA, with two differences. First, whenever an integer value n participates in an operation as belonging to a data type int_π, any other integer n' can be used nondeterministically so long as $n' \approx_\pi n$.

$$\frac{N_1, s_1 \Longrightarrow n_1, s_2 \qquad N_2, s_2 \Longrightarrow n_2, s}{\mathsf{op}_{\mathsf{int}_{\pi_1} \to \mathsf{int}_{\pi_2} \to \mathsf{int}_\pi}\, N_1\, N_2, s_1 \Longrightarrow n', s}\ n_i' \approx_{\pi_i} n_i,\ i = 1, 2,\ n' \approx_\pi \mathsf{op}\, n_1'\, n_2'$$

Assignment and dereferencing have similar nondeterministic rules.

Second, the abort program with any state reduces to $\mathcal{E}$, and a composite program reduces to $\mathcal{E}$ if a subprogram is reduced and results in $\mathcal{E}$. For any language operator op

$$\mathsf{abort}, s \Longrightarrow \mathcal{E} \qquad \frac{M, s \Longrightarrow \mathcal{E}}{\mathsf{op}\, M\; M' \Longrightarrow \mathcal{E}} \ ,$$

2.1 Observational Safety

A program M is said to *terminate* in state s if there exists configuration $\mathcal{K}$ such that $\mathcal{K} = \mathcal{E}$ or $\mathcal{K} = \mathsf{skip}, s'$ for some state s' such that $M, s \Longrightarrow \mathcal{K}$. If $\mathcal{K} \neq \mathcal{E}$ we say M is *safe*. Term $\Gamma \vdash M : \theta$ *approximates* term $\Gamma \vdash M' : \theta$, denoted by $\Gamma \vdash M \sqsubsetsim M'$ if for all contexts $\mathcal{C}[-]$, the termination of program $\mathcal{C}[M]$ implies the termination of program $\mathcal{C}[M']$. If two terms approximate each other they are considered *equivalent*, denoted by $\Gamma \vdash M \cong M'$.

A context is safe if it does not include occurrences of the abort command. A term M is *safe* if for any safe context $\mathcal{C}_{\text{safe}}[-]$ program $\mathcal{C}_{\text{safe}}[M]$ is safe; otherwise the term is *unsafe*.

3 Game Semantics of AIA

Game semantics emerged in the last decade as a potent framework for modeling programming languages [2, 17, 3, 18, 11, 15]. It is an alternative (to Scott-Strachey) denotational semantics which interprets types as *arenas* (i.e. structured sets of atomic *moves*), computation as *plays* (i.e. structured sequences of moves) and terms as *strategies* (i.e. structured sets of plays). Strategies compose, much like CSP-style processes, which makes it possible to define denotational models. For technical details, the reader is referred to loc. cit.

Except for the presence of abort, AIA is syntactic sugar on top of IA with Erratic choice (EIA). We will use the may-termination model presented in [14, Chap. 3]. For any integer abstraction π, let $\mathsf{blur}_{\mathsf{expint}_\pi} : \mathsf{expint} \to \mathsf{expint}$ denote an EIA term which, given an integer n, returns a nondeterministically chosen integer n' such that $n' \approx_\pi n$.[2] For all other AIA types θ, we define EIA terms $\mathsf{blur}_\theta : \widetilde{\theta} \to \widetilde{\theta}$ as follows:

$$\begin{array}{l}
\mathsf{blur}_{\mathsf{expbool}} = \lambda\, x : \mathsf{expbool}.x \qquad \mathsf{blur}_{\mathsf{com}} = \lambda\, x : \mathsf{com}.x \\
\mathsf{blur}_{\mathsf{var}\tau} = \lambda\, x : \mathsf{var}\widetilde{\tau}.\mathsf{mkvar}\, (\lambda\, y : \mathsf{exp}\widetilde{\tau}.\mathsf{asg}\, x\, (\mathsf{blur}_{\mathsf{exp}\tau}\, y))\, (\mathsf{blur}_{\mathsf{exp}\tau}(\mathsf{der} x)) \\
\mathsf{blur}_{\theta \to \theta'} = \lambda f : \widetilde{\theta} \to \widetilde{\theta}'.\, \lambda\, x : \widetilde{\theta}.\mathsf{blur}_{\theta'}(f(\mathsf{blur}_\theta x))
\end{array}$$

For any AIA type θ, its translation $\ulcorner \theta \urcorner$ into EIA is $\widetilde{\theta}$. The translation of any AIA term into EIA is defined by:

[2] Since abstractions are assumed computable, such terms are definable in EIA by iteratively testing all integers n'. However, in addition to the possibilities to choose any n' with $n' \approx_\pi n$ nondeterministically, there is the possibility of divergence. Therefore, this approach works only for may-termination semantics, which is sufficient in this paper.

$$\ulcorner \mathsf{k} : \theta \urcorner = \mathsf{blur}_\theta \mathsf{k} : \widetilde{\theta} \qquad \ulcorner M\,N : \theta \urcorner = \ulcorner M : \theta_1 \to \theta \urcorner \ulcorner N : \theta_2 \urcorner$$
$$\ulcorner x : \theta \urcorner = \mathsf{blur}_\theta x : \widetilde{\theta} \qquad \ulcorner \lambda\, x : \theta . M : \theta \to \theta' \urcorner = \lambda\, x : \widetilde{\theta} . \ulcorner M : \theta' \urcorner$$

The semantic model of AIA is therefore essentially that of EIA, which is presented in detail in [15]. Below, we give a sketch of the model.

An *arena* A is a triple $\langle M_A, \lambda_A, \vdash_A \rangle$ where M_A is a set of *moves*, $\lambda_A : M_A \to \{O, P\} \times \{Q, A\}$ is a function determining for each $m \in M_A$ whether it is an *Opponent* or a *Proponent move*, and a *question* or an *answer*. We write $\lambda_A^{OP}, \lambda_A^{QA}$ for the composite of λ_A with respectively the first and second projections. $\vdash_A$ is a binary relation on M_A, called *enabling*, satisfying: if $m \vdash_A n$ for no m then $\lambda_A(n) = (O, Q)$, if $m \vdash_A n$ then $\lambda_A^{OP}(m) \neq \lambda_A^{OP}(n)$, and if $m \vdash_A n$ then $\lambda_A^{QA}(m) = Q$. If $m \vdash_A n$ we say that m *enables* n. We shall write I_A for the set of all moves of A which have no enabler; such moves are called *initial*. Note that an initial move must be an Opponent question.

An arena is called *flat* if its questions are all initial (consequently the P-moves can only be answers). Flat arenas interpret base types, and are determined by their enabling relations:

$$\begin{aligned} \mathsf{com} : &\quad run \vdash done, abort \\ \mathsf{exp}\tau : &\quad q \vdash n, abort \\ \mathsf{var}\tau : &\quad read \vdash n, abort, \quad write(n) \vdash ok, abort. \end{aligned}$$

The *product* $(A \times B)$ and *arrow* $(A \Rightarrow B)$ arenas are defined by:

$$\begin{aligned} M_{A\times B} &= M_A + M_B & M_{A\Rightarrow B} &= M_A + M_B \\ \lambda_{A\times B} &= [\lambda_A, \lambda_B] & \lambda_{A\Rightarrow B} &= [\langle \lambda_A^{PO}, \lambda_A^{QA} \rangle, \lambda_B] \\ \vdash_{A\times B} &= \vdash_A + \vdash_B & \vdash_{A\Rightarrow B} &= \vdash_A \cup \vdash_B \cup (I_B \times I_A) \end{aligned}$$

where $\lambda_A^{PO}(m) = O$ iff $\lambda_A^{OP}(m) = P$.

A *justified sequence* in arena A is a finite sequence of moves of A equipped with pointers. The first move is initial and has no pointer, but each subsequent move n must have a unique pointer to an earlier occurrence of a move m such that $m \vdash_A n$. We say that n is (explicitly) justified by m or, when n is an answer, that n answers m. A *legal play* is a justified sequence with some additional constraints. *Alternation* and *well-threaded-ness* are standard in game semantics, to which we add the following:

Definition 1 (Halting plays). *No moves can follow* abort*.*

This represents the abrupt termination caused by aborting. The set of all legal plays in arena A is denoted by P_A.

A *strategy* is a prefix-closed set of even length plays. Strategies compose in a way which is reminiscent of parallel composition plus hiding in process calculi. We call a play *complete* if either the opening question is answered or the special move *abort* has been played.

Two strategies $\sigma : A \Rightarrow B'$ and $\tau : B'' \Rightarrow C$ can be composed by considering their possible interactions in the shared arena B (the decorations are only used

to distinguish the two occurrences of this type). Moves in B are subsequently hidden yielding a sequence of moves in A and C.

Let u be a sequence of moves from arenas A, B', B'' and C with justification pointers from all moves except those initial in C, such that pointers from moves in C cannot point to moves in A and vice versa. Define $u \upharpoonright B'', C$ to be the subsequence of u consisting of all moves from B'' and C (pointers between A-moves and B''-moves are ignored). $u \upharpoonright A, B'$ is defined analogously (pointers between B' and C are then ignored).

Definition 2 (Interaction sequence). *We say that justified sequence u is an* interaction sequence *of A, B', B'' and C if:*

1. *any move from $I_{B''}$ is followed by its copy in $I_{B'}$,*
2. *any answer to a move in $I_{B'}$ is followed by its copy in $I_{B''}$*
3. *$u \upharpoonright A, B' \in P_{A \Rightarrow B'}$, $u \upharpoonright A, C \in P_{A \Rightarrow C}$, $u \upharpoonright B'', C \in P_{B'' \Rightarrow C}$.*

The set of all such sequences is written as $int(A, B, C)$. Composing the two strategies σ and τ yields the following set of interaction sequences:

$$\sigma \between \tau = \{u \in int(A, B, C) \mid u \upharpoonright A, B' \in \sigma,\ u \upharpoonright B'', C \in \tau\}$$

Suppose $u \in int(A, B, C)$. Define $u \upharpoonright A, C$ to be the subsequence of u consisting of all moves from A and C, but where there was a pointer from a move $m_A \in M_A$ to an initial move $m \in I_{B''}$ extend the pointer to the initial move in C which was pointed to from its copy $m_{B'}$. The strategy which is the composition of σ and τ is then defined as $\sigma;\ \tau = \{u \upharpoonright A, C \mid u \in \sigma \between \tau\}$.

Strategies are used to give denotations to terms. Language constants, including functional constants, are interpreted by strategies and terms are constructed using strategy composition. Lambda abstraction and currying are isomorphisms consisting only of re-tagging of move occurrences. We interpret abort using the strategy $[\![\mathsf{abort}]\!] = \{\epsilon,\ run \cdot abort\}$.

3.1 Full Abstraction

Using standard game-semantic techniques we can show that the above model is fully abstract for AIA.

Theorem 1 (Full abstraction). *For any terms $\Gamma \vdash M, M' : \theta$, $\Gamma \vdash M \sqsubseteq M'$ iff $[\![\Gamma \vdash M : \theta]\!] \subseteq [\![\Gamma \vdash M' : \theta]\!]$.*

Proof (sketch). The proof follows the pattern of [14, Sec. 3.8]. In the presence of abort it is no longer necessary to use quotienting on strategies, as they are characterised by their full set of plays. The proof of this property is similar to that of the Characterisation Theorem for IA [3, Thm. 25]. The basic idea is that we can interrupt any play (not necessarily complete) by composing it with a stateful strategy that plays *abort* at the right moment. □

Note that in the presence of abort it is no longer the case that strategies are characterised by their set of complete plays, as it is the case for EIA. This is

consistent with the fact that terms such as $c : \mathsf{com} \vdash c; \mathsf{diverge}$ and $\mathsf{diverge}$ are no longer equivalent although they both have same set of complete plays (empty). Command c may cause abort, thus preventing divergence. This is a common property of languages with control [18], and abort is such a feature.

Let us call a play *safe* if it does not terminate in *abort*, and a strategy if it consists only of safe plays; otherwise, we will call plays and strategies *unsafe*. From the full abstraction result it follows that:

Corollary 1 (Safety). *$\Gamma \vdash M : \theta$ is safe iff $[\![\Gamma \vdash M : \theta]\!]$ is safe.*

This result ensures that model-checking a strategy for safety (i.e. the absence of the *abort* move) is equivalent to proving the safety of a term.

3.2 Quotient Semantics

Given a base type expint_π or varint_π of AIA, we can quotient the arena and game for expint or varint (respectively) in a standard way, by replacing any integer n with its equivalence class $\{m \mid m \approx_\pi n\}$. This extends compositionally to any type θ of AIA: we can quotient the arena and game for $\widetilde{\theta}$ by the abstractions in θ. For any play t of the game for $\widetilde{\theta}$, let $\overline{t}$ denote the image play of the quotient game, obtained by replacing each integer in t by its equivalence class in the corresponding abstraction in θ.

It is straightforward to check that, for any term $\Gamma \vdash M : \theta$ of AIA, and plays t and t' of the game for $\widetilde{\Gamma} \vdash \widetilde{\theta}$, such that $\overline{t} = \overline{t'}$, we have

$$t \in [\![\Gamma \vdash M : \theta]\!] \;\Leftrightarrow\; t' \in [\![\Gamma \vdash M : \theta]\!]$$

Therefore, the quotient of the strategy $[\![\Gamma \vdash M : \theta]\!]$ by the abstractions in Γ and θ loses no information.

Moreover, the quotient strategies can be defined compositionally, i.e. by recursion on the typing rules of AIA. The most interesting case is functional application $\Gamma \vdash M\,N : \theta$, where $\Gamma \vdash M : \theta_1 \to \theta$, $\Gamma \vdash N : \theta_2$, and $\widetilde{\theta_1} = \widetilde{\theta_2}$. Since the abstractions in θ_1 and θ_2 may be different, we need to allow a move which contains an equivalence class c to interact with any move obtained by replacing c with some c' such that $c \cap c' \neq \emptyset$. Hence, even if the quotient strategies for M and N are deterministic, the one for $M\,N$ may be nondeterministic.

In the rest of the paper, $[\![\Gamma \vdash M : \theta]\!]$ will denote the quotient strategy.

Example 1. Consider the quotient strategy

$$[\![x : \mathsf{varint}_{[0,4]} \vdash x := !x +_{[0,4]\to[0,1]\to[0,3]} 1_{[0,1]} : \mathsf{com}]\!]$$

If the abstract value (i.e. equivalence class) 3 is read from the variable x, the result of the addition is >3, because it belongs to the abstraction $[0,3]$. When >3 is assigned to x which is abstracted by $[0,4]$, it is nondeterministically converted to either 4 or >4. Thus, the following are two possible complete plays:

$$run\, read_x\, 3_x\, write(4)_x\, ok_x\, ok, \quad run\, read_x\, 3_x\, write(>4)_x\, ok_x\, ok$$

3.3 Interaction Semantics

In *standard* semantics, which is presented above, to obtain the strategy $[\![\Gamma \vdash M\,N : \theta]\!]$, the strategies $[\![\Gamma \vdash M : \theta_1 \rightarrow \theta]\!]$ and $[\![\Gamma \vdash N : \theta_2]\!]$ are composed, and moves which interact are hidden. (Here $\widetilde{\theta_1} = \widetilde{\theta_2}$.)

Let $\langle\!\langle - \rangle\!\rangle$ denote an alternative semantics, where moves which interact are not hidden. We call this the *interaction* semantics, and its building blocks interaction plays and interaction strategies.

For any term $\Gamma \vdash M : \theta$ of AIA, its interaction semantics can be easily reconciled with its standard semantics, by performing all the hiding at once. In the following, $- \upharpoonright \Gamma, \theta$ indicates restriction to the arenas corresponding to base types occurring in Γ and θ.

Proposition 1. *$[\![\Gamma \vdash M : \theta]\!] = \langle\!\langle \Gamma \vdash M : \theta \rangle\!\rangle \upharpoonright \Gamma, \theta$.*

Standard plays are alternating sequences of Opponent and Player moves. Interaction plays in addition contain internal moves, which do not interact in subsequent compositions, but which record all intermediate steps taken during the computation.

Consider composing $\langle\!\langle \Gamma \vdash M : \theta_1 \rightarrow \theta \rangle\!\rangle$ and $\langle\!\langle \Gamma \vdash N : \theta_2 \rangle\!\rangle$ to obtain $\langle\!\langle \Gamma \vdash M\,N : \theta \rangle\!\rangle$. According to the definition of interaction sequences above, for any moves r_1 and r_2 whose types σ_1 and σ_2 (respectively) are corresponding base types in θ_1 and θ_2, and which interact, they are both recorded in $\langle\!\langle \Gamma \vdash M\,N : \theta \rangle\!\rangle$. Indeed, since we only have $\widetilde{\theta_1} = \widetilde{\theta_2}$, r_1 and r_2 may be different. However, if σ_1 and σ_2 are not types of integer expressions or integer variables, then $\sigma_1 = \sigma_2$ and $r_1 = r_2$. In such cases, when presenting interaction plays and strategies, we may record r_1 and r_2 only once, for readability.

Example 2. Consider the interaction strategy of the term in Example 1. Here is one of its complete interaction plays, corresponding to the second standard play in Example 1. Any internal move is tagged with the coordinates of the corresponding sub-term. For instance, $q_{2,1}$ is the question to the sub-term $!x$, which is the 1st immediate sub-term of $!x + 1$, which in turn is the 2nd immediate sub-term of $x := !x + 1$. Observe also the double occurrences of integer internal moves, in line with how interaction plays are composed. In this example, those pairs are equal because, in any functional application, any two corresponding abstractions are equal. An abstract value needs to be converted to another abstraction only within the strategy for asg, where a value with abstraction $[0, 3]$ is assigned to a variable with abstraction $[0, 4]$.

$$\begin{aligned} &\mathit{run}\; q_2\; q_{2,1}\; q_{2,1,1}\; \mathit{read}_x\; 3_x\; 3_{2,1,1}\; 3_{2,1,1}\; 3_{2,1}\; 3_{2,1}\; q_{2,2}\; 1_{2,2}\; 1_{2,2} \\ &\qquad\qquad (>3)_2\; (>3)_2\; \mathit{write}(>4)_1\; \mathit{write}(>4)_x\; \mathit{ok}_x\; \mathit{ok}_1\; \mathit{ok} \end{aligned}$$

The interaction semantics, rather than the standard semantics, will be used for the purpose of abstraction refinement. The reason is that, given an unsafe standard play of an abstracted term, it does not in general contain sufficient information to decide that it can be produced by the concrete version of the

term (i.e. that it is not a *spurious counterexample*), or to choose one or more abstractions to be refined for the next iteration.

In classical, stateful, abstraction-refinement an abstract counterexample to a safety property is guaranteed to be genuine if the computation was deterministic (or, at least, the nondeterminism was not caused by over-abstraction). In standard semantics, however, all internal steps within a computation are hidden. This results in standard strategies of abstracted terms in general not containing all information about sources of their nondeterminism.

Example 3. Consider the following abstracted term, with notation in Fig. 1:

$$\vdash \mathsf{newint}_{[]}\; x := 0_{[0,0]} \;\mathsf{in}\;\mathsf{if}\;(x \neq 0_{[0,0]})\;\mathsf{abort}\,\mathsf{skip} : \mathsf{com}$$

Its complete standard plays are *run abort* and *run ok*. In fact, its strategy is the same as the strategy of the EIA term abort or skip. However, the counterexample *run abort* is spurious, and the abstraction of x needs to be refined, but internal moves which point to this abstraction as the source of nondeterminism have been hidden.

4 Conservativity of Abstraction

As interaction plays contain internal moves, we can distinguish those whose underlying computation did not pass through any nondeterministic branching that is due to abstraction.

Definition 3. *(a) Given integer abstractions π and π', and an abstract value (i.e. equivalence class) c of π, we say that converting c to π' is* deterministic *if there exists an abstract value c' of π' such that $c \subseteq c'$.*

(b) Given an abstracted operation $\mathsf{op} : \mathsf{exp}\tau_1 \rightarrow \mathsf{exp}\tau_2 \rightarrow \mathsf{exp}\tau$ and abstract values c_1 and c_2 of type τ_1 and τ_2 respectively, we say that the application of op to c_1 and c_2 is deterministic *if there exists an abstract value c of type τ such that $\forall\, v_1 \in c_1, v_2 \in c_2,\; \mathsf{op}\, v_1\, v_2 \;\in\; c$.*[3]

(c) An interaction play $u \in \langle\!\langle \Gamma \vdash M : \theta \rangle\!\rangle$ is deterministic *if each conversion of an abstract integer value in u is deterministic, and each application of an arithmetic-logic operator in u is deterministic.*

For abstractions π and π', we say that π' *refines* π if, for any equivalence class c' of π', there exists an equivalence class c of π such that $c' \subseteq c$. When π' refines π, and c is an equivalence class of π, we say that π' *splits* c if c is not an equivalence class of π'.

We say that a term $\Gamma' \vdash M' : \theta'$ *refines* a term $\Gamma \vdash M : \theta$ if $\widetilde{\Gamma'} = \widetilde{\Gamma}$, $\widetilde{M'} = \widetilde{M}$, $\widetilde{\theta'} = \widetilde{\theta}$, and each abstraction in $\Gamma' \vdash M' : \theta'$ refines the corresponding abstraction in $\Gamma \vdash M : \theta$.

Theorem 2. *Suppose $\Gamma' \vdash M' : \theta'$ refines $\Gamma \vdash M : \theta$.*

[3] Here we regard the abstract values tt and ff as singleton sets $\{tt\}$ and $\{ff\}$.

(i) *For any $t \in [\![\Gamma' \vdash M' : \theta']\!]$, we have $\overline{t} \in [\![\Gamma \vdash M : \theta]\!]$. The same is true for the $\langle\!\langle - \rangle\!\rangle$ semantics.*
(ii) *For any deterministic $u \in \langle\!\langle \Gamma \vdash M : \theta \rangle\!\rangle$, there exists $t \in \langle\!\langle \Gamma' \vdash M' : \theta' \rangle\!\rangle$ such that $u = \overline{t}$.*[4]

Proof. By induction on the typing rules of AIA. □

The following consequence of Corollary 1, Proposition 1 and Theorem 2 will justify the correctness of the abstraction refinement procedure.

Corollary 2. *Suppose $\Gamma' \vdash M' : \theta'$ refines $\Gamma \vdash M : \theta$.*

(i) *If $[\![\Gamma \vdash M : \theta]\!]$ is safe, then $\Gamma' \vdash M' : \theta'$ is safe.*
(ii) *If $\langle\!\langle \Gamma \vdash M : \theta \rangle\!\rangle$ has a deterministic unsafe interaction play, then $\Gamma' \vdash M' : \theta'$ is unsafe.*

5 Abstraction Refinement

In the rest of the paper, we work with the 2nd-order recursion-free fragment of AIA. In particular, function types are restricted to $\theta ::= \sigma \mid \sigma \rightarrow \theta$. Instead of the functional constant new is more convenient to use the combinator $\mathsf{new}\tau\, x := E \;\mathsf{in}\; M$ which binds free occurrences of x in M. Without loss of generality, we consider only normal forms with respect to β-reduction.

An abstraction π is *finitary* if it has finitely many equivalence classes. A term is *finitely abstracted* if it contains only finitary abstractions.

A set of abstractions is *effective* if their equivalence classes have finite representations, and if conversions of abstract values between abstractions, and all arithmetic-logic operators over abstract values, are computable.

Proposition 2. *For any finitely abstracted term $\Gamma \vdash M : \theta$ with abstractions from an effective set, the set $[\![\Gamma \vdash M : \theta]\!]$ is a regular language. Moreover, an automaton which recognises it is effectively constructible. The same is true for the $\langle\!\langle - \rangle\!\rangle$ semantics.*

Proof. Since the abstractions are finitary, $\Gamma \vdash M : \theta$ can be seen as a term of 2nd-order recursion free EIA with finite data types and abort. We can extend the construction in [10] to obtain effectively an automaton which recognises $[\![\Gamma \vdash M : \theta]\!]$. Note that the construction in loc. cit. characterises strategies in terms of their *complete plays*, i.e. those plays in which the initial question is answered. However, in the presence of abort strategies are defined by their full sets of plays is (Theorem 1), so to each finite state machine used in loc. cit. we apply a suitable prefix closure operator, which preserves the finite-state property.

To obtain an automaton for $\langle\!\langle \Gamma \vdash M : \theta \rangle\!\rangle$, interacting moves are tagged with sub-term coordinates rather than hidden. □

[4] This can be strengthened to apply to interaction plays which are deterministic with respect to the abstractions in $\Gamma' \vdash M' : \theta'$. The latter notion allows nondeterministic conversions of, and operator applications to, abstract values which are not split by the corresponding abstractions in $\Gamma' \vdash M' : \theta'$.

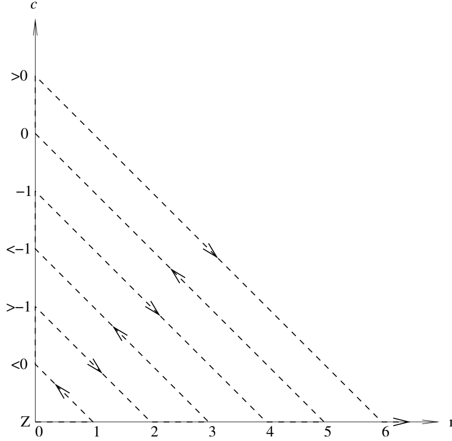


Fig. 2. A possible definition of $\sqsubseteq$

Let $A[\![\Gamma \vdash M : \theta]\!]$ and $A\langle\!\langle \Gamma \vdash M : \theta \rangle\!\rangle$ denote the automata obtained as in the proof of Proposition 2. Since there is no hiding in the construction of $A\langle\!\langle \Gamma \vdash M : \theta \rangle\!\rangle$, this automaton is deterministic.

Given a finite word u and a deterministic automaton A which accepts u, we call u *cycle-free* if the accepting run visits any state of A at most once.

Apart from the identity abstraction κ, for simplicity, from now on we work only with the abstractions $[]$ and $[n, m]$, where $n \leq 0 \leq m + 1$ (see Fig. 1). Observe that these abstractions are finitary and form an effective set.

Let $\prec$ denote the following computable linear ordering between abstract values:

$$\mathbb{Z} \prec ({<}0) \prec ({>}{-1}) \prec ({<}{-1}) \prec -1 \prec 0 \prec ({>}0) \prec \cdots \\ ({<}{-(n+1)}) \prec -(n+1) \prec n \prec ({>}n) \prec \cdots$$

For two moves (possibly tagged with sub-term coordinates) r and r' which are equal except for containing different abstract integer values c and c', let $r \prec r'$ if $c \prec c'$, and $r' \prec r$ if $c' \prec c$. Now, we extend this ordering to a computable linear ordering on all moves (in an arbitrary but fixed way), and denote it by $\prec$. Let $\prec$ also denote the linear orderings on plays obtained by lifting the linear ordering on moves lexicographically.

Let $(n, c) \sqsubseteq (n', c')$ be any computable linear ordering between pairs of non-negative integers and abstract integer values which is obtained by extending the partial ordering defined by $n \leq n'$ and $c \preceq c'$, and which admits no infinite strictly decreasing sequences, and no infinite strictly increasing sequences bounded above (see Fig. 2). For any play u, let $|u|$ denote its length, and $\max(u)$ denote the $\prec$-maximal abstract integer value in u (or $\mathbb{Z}$ if there is no such value). Let $u \sqsubseteq u'$ mean $(|u|, \max(u)) \sqsubseteq (|u'|, \max(u'))$. Now, let $\trianglelefteq$ be the linear order-

The procedure checks safety of a given concrete term $\Gamma \vdash M : \theta$.

1 Let $\Gamma_0 \vdash M_0 : \theta_0$ be a finitely abstracted anti-refinement of $\Gamma \vdash M : \theta$, i.e. be obtained from $\Gamma \vdash M : \theta$ by replacing κ by finitary abstractions. Let $i := 0$.
2 If $A[\![\Gamma_i \vdash M_i : \theta_i]\!]$ accepts only safe plays, terminate with answer SAFE.
3 Otherwise, if $A\langle\!\langle \Gamma_i \vdash M_i : \theta_i \rangle\!\rangle$ accepts a deterministic unsafe interaction play, terminate with answer UNSAFE.
4 Otherwise, let u be the $\trianglelefteq$-minimal unsafe interaction play accepted by $A\langle\!\langle \Gamma_i \vdash M_i : \theta_i \rangle\!\rangle$. Let $\Gamma_{i+1} \vdash M_{i+1} : \theta_{i+1}$ be obtained by refining one or more abstractions in $\Gamma_i \vdash M_i : \theta_i$ by finitary abstractions, provided that at least one abstract value which occurs in u is split. Let $i := i + 1$, and repeat from **2**.

Fig. 3. Abstraction refinement procedure

ing between plays such that $u \trianglelefteq u'$ if and only if either $u \sqsubset u'$, or $|u| = |u'|$, $\max(u) = \max(u')$ and $u \preceq u'$.

Lemma 1. *In the linear order of all plays with respect to $\trianglelefteq$:*

(i) there is no infinite strictly decreasing sequence;
(ii) there is no infinite strictly increasing sequence which is bounded above.

Proof. This is due to the following two facts. Firstly, the $\sqsubseteq$ ordering between pairs of nonnegative integers and abstract integer values has the properties (i) and (ii). Secondly, for any such pair (n, c), there are only finitely many plays u such that $|u| = n$ and $\max(u) = c$. $\square$

The abstraction refinement procedure (ARP) is given in Fig. 3. Note that, in step **1**, the initial abstractions can be chosen arbitrarily; and in step **4**, arbitrary abstractions can be refined in arbitrary ways, as long as that splits at least one abstract value in u. These do not affect correctness and semi-termination, but they allow experimentation with different heuristics in concrete implementations.

Theorem 3. *ARP is well-defined and effective. If it terminates with SAFE (UNSAFE, respectively), then $\Gamma \vdash M : \theta$ is safe (unsafe, respectively).*

Proof. For well-defined-ness, Lemma 1 (i) ensures that the $\trianglelefteq$-minimal unsafe interaction play u accepted by $A\langle\!\langle \Gamma_i \vdash M_i : \theta_i \rangle\!\rangle$ always exists. Since the condition in step **3** was not satisfied, u is not deterministic. Therefore, u cannot contain only singleton abstract values, so there is at least one abstract value in u which can be split.

Effectiveness follows from Proposition 2, by the fact that it suffices to consider cycle-free plays in step **4**, and from computability of $\trianglelefteq$.

If ARP terminates with SAFE (UNSAFE, respectively), then $\Gamma \vdash M : \theta$ is safe (unsafe, respectively) by Corollary 2, since any abstraction is refined by the identity abstraction κ. $\square$

Theorem 4. *If $\Gamma \vdash M : \theta$ is unsafe then ARP will terminate with UNSAFE.*

Proof. By Corollary 1 and Proposition 1, there exists an unsafe $t \in \langle\langle \Gamma \vdash M : \theta \rangle\rangle$.

For each i, let U_i be the set of all unsafe $u \in \langle\langle \Gamma_i \vdash M_i : \theta_i \rangle\rangle$, and let $u_i^\dagger$ be the $\trianglelefteq$-minimal element of U_i.

It follows by Theorem 2 that, for any $u \in \langle\langle \Gamma_{i+1} \vdash M_{i+1} : \theta_{i+1} \rangle\rangle$, $\overline{u} \in \langle\langle \Gamma_i \vdash M_i : \theta_i \rangle\rangle$. Also, we have $\overline{u} \trianglelefteq u$. Now, step **4** ensures that, for any i, $u_i^\dagger \notin \langle\langle \Gamma_{i+1} \vdash M_{i+1} : \theta_{i+1} \rangle\rangle$.

Therefore, $u_0^\dagger \lhd u_1^\dagger \lhd \cdots u_i^\dagger \lhd \cdots$. But, for each i, $u_i^\dagger \trianglelefteq \overline{t}^i \trianglelefteq t$. By Lemma 1 (ii), ARP must terminate for $\Gamma \vdash M : \theta$! □

ARP may diverge for safe terms. This is generally the case with abstraction refinement methods since the underlying problem is undecidable. A simple example is the term

$$e : \mathsf{expint} \vdash \mathsf{newint}\, x := e \,\mathsf{in}\, \mathsf{if}\, (!x = !x + 1)\, \mathsf{abort}\, \mathsf{skip} : \mathsf{com}$$

This term is safe, but any finitely abstracted anti-refinement of it is unsafe.

6 Conclusions and Related Work

In this paper, we extended the applicability of game-based software model checking by a data-abstraction refinement procedure which applies to open program fragments which can contain infinite integer types, and which is guaranteed to discover an error if it exists. The procedure is made possible and it was justified by a firm theoretical framework. Some interesting topics for future work are dealing with terms which contain recursion, and extending to a concurrent programming language [12] or higher-order fragments [22].

The pioneering applications of game models to program analysis were by Hankin and Malacaria [13, 19–21], who also use nondeterminism as a form of abstraction. Their abstraction techniques apply to higher-order constructs rather than just data, by forgetting certain information used in constructing the game models (the *justification pointers*). It is an interesting question whether this style of abstraction can be iteratively refined. The first applications of game-semantic models to model checking were by Ghica and McCusker [10]. The latter line of research was further pursued as part of the *Algorithmic Game Semantics* research programme at the University of Oxford [1], and by Dimovski and Lazić [9].

On the topics of data abstraction [7] and abstraction refinement [6], there is a literature too vast to mention. Good entry points, which also represented essential motivation for our work, are the articles written on the SLAM model-checker [4]. It is too early to compare our approach with traditional, stateful, model-checkers. The first obstacle is the use of different target languages to express programs, but we hope to move towards more realistic target languages in the near future. The second obstacle stems from a difference of focus. Stateful techniques are already very mature and can target realistic industrial software; their overriding concern is efficiency. Our main concern, on the other hand, is *compositionality*, which we believe can be achieved in a clean and theoretically solid way by using a semantics-directed approach. In order to narrow the gap

between the efficiency of stateful tools and game-based tools, many program analysis techniques need to be re-cast using this new framework. Judging by the positive initial results, we trust the effort is worthwhile. Compositionality is a worthwhile long-term goal as compositional techniques are the best guarantee of scalability to large systems.

Aside from compositionality, one important advantage of game-based models is their small size, which is achieved by hiding all unobservable internal actions. However, in order to identify and analyse counterexample traces it is necessary, as we have pointed out in Sec. 3.3, to expose internal actions. In order to implement this abstraction refinement procedure reasonably, we must proceed by first identifying counter-example *standard* plays, and then obtaining corresponding *interaction* plays by "uncovering" the hidden moves. We are currently developing a model-checking tool based on representing strategies in the process algebra CSP [23], which can be verified using the FDR model checker. We can exploit a feature of FDR which allows identification of hidden events in counterexample traces, in order to implement the "uncovering" operation necessary to compute interaction plays efficiently.

References

1. S. Abramsky, D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Applying game semantics to compositional software modeling and verification. In Proceedings of *TACAS*, LNCS **2988**, (2004), 421–435.
2. S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, **163(2)**, (2000).
3. S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In P.W.O'Hearn and R.D.Tennent, editors, *Algol-like languages*. (Birkhaüser, 1997).
4. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In Proceedings of *TACAS*, LNCS **2280**, (2002), 158–172.
5. T. Ball and S. K. Rajamani. Debugging System Software via Static Analysis. n Proceedings of *POPL*, ACM SIGPLAN Notices **37**(1), (2002), 1–3.
6. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In Proceeding of *CAV*, LNCS **1855**, (2000), 154–169.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of *POPL*, (1977), 238–252.
8. P. Cousot and R. Cousot. Modular static program analysis. In Proceedings of *CC*, (2002).
9. A. Dimovski and R. Lazic. Csp representation of game semantics for second-order idealized algol. In Proceedings of *ICFEM*, LNCS **3308**, (2004), 146–161.
10. D. R. Ghica and G. McCusker. The Regular-Language Semantics of Second-order Idealized Algol. Theoretical Computer Science **309** (1–3), (2003), 469–502.
11. D. R. Ghica and A. S. Murawski. Angelic semantics of fine-grained concurrency. In Proceedings of *FoSSaCS*, LNCS **2987**, (2004), 211-255.
12. D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Syntactic control of concurrency. In Proceedings of *ICALP*, LNCS **3142**, (2004).

13. C. Hankin and P. Malacaria. Program analysis games. *ACM Comput. Surv.*, 31(3es), (1999).
14. R. Harmer Games and Full Abstraction for Nondeterministic Languages. Ph. D. Thesis Imperial College, 1999.
15. R. Harmer and G. McCusker. A fully abstract game semantics for finite nondeterminism. In Proceedings of*LICS*, (1999).
16. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In Proceedings of *SPIN*, (2003).
17. J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Information and Computation* **163**, (2000), 285–400.
18. J. Laird. A fully abstract game semantics of local exceptions. In Proceedings of *LICS*, (2001).
19. P. Malacaria and C. Hankin. Generalised flowcharts and games. In Proceedings of *ICALP*, (1998).
20. P. Malacaria and C. Hankin. A new approach to control flow analysis. In Proceedings of *CC*, (1998).
21. P. Malacaria and C. Hankin. Non-deterministic games and program analysis: An application to security. In Proceedings of *LICS*, (1999).
22. A. Murawski and I. Walukiewicz Third-Order Idealized Algol with Iteration Is Decidable. In Proceedings of *FoSSaCS*, LNCS **3411**, (2005), 202–218.
23. W. A. Roscoe. *Theory and Practice of Concurrency.* Prentice-Hall, 1998.

Locality-Based Abstractions

Javier Esparza[1], Pierre Ganty[2,⋆], and Stefan Schwoon[1]

[1] Institut für Formale Methoden der Informatik, Universität Stuttgart
{esparza, schwoosn}@informatik.uni-stuttgart.de
[2] Département d'Informatique, Université Libre de Bruxelles
pganty@ulb.ac.be

Abstract. We present locality-based abstractions, in which a set of states of a distributed system is abstracted to the collection of views that some observers have of the states. Special cases of locality-abstractions have been used in different contexts (planning, analysis of concurrent programs, concurrency theory). In this paper we give a general definition in the context of abstract interpretation, show that arbitrary locality-based abstractions are hard to compute in general, and provide two solutions to this problem. The solutions are evaluated in several case studies.

1 Introduction

Consider a system acting on a set X of program variables over some value set V. An abstraction of the system, in the abstract-interpretation sense [1], deliberately loses information about the current values of the variables. Many abstractions can be intuitively visualized by imagining an observer who has access to the program code but is only allowed to retain limited knowledge about the values of the variables. For instance, the observer may only be allowed to retain the sign of a variable, its value modulo a number, or whether one value is larger than another one. In this paper we consider *locality-based* abstractions, which are best visualized by imagining a set of observers, each of which has a partial view of the system. Each observer has access to all the information 'within his window', but no information outside of it. For instance, in a system with three variables there could be three observers, each of them with perfect information about two of the variables, but no knowledge about the third. Given the set $\{\langle 1,1,0\rangle, \langle 1,0,1\rangle, \langle 0,1,1\rangle\}$ of valuations of the variables, the observer with access to, say, the first two variables 'sees' $\{\langle 1,1,\mathbf{u}\rangle, \langle 1,0,\mathbf{u}\rangle, \langle 0,1,\mathbf{u}\rangle\}$, where $\mathbf{u}$ stands for absence of information. Notice that information is lost: Even if the three observers exchange their informations, they cannot conclude that $\langle 1,1,1\rangle$ does *not* belong to the set of valuations.

The idea of local observers is particularly appropriate for distributed systems in which the value of a variable corresponds to the *local state* of a component of the system. In this case, a partial view corresponds to having no information from a number of components of the system. This is also the reason for the term "locality-based" abstraction.

⋆ The author wishes to thank the University of Stuttgart, where most of the work was done, for hospitality and both FRIA and FNRS for financial support.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 118–134, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Plan of the paper. In Sect. 3 we present a very general definition of locality-based abstraction, and study, in Sect. 4, the problem of computing the abstract $post^{\#}$ operator (i.e., the abstract operator corresponding to the usual *post* operator that computes the set of immediate successors on the concrete space). We observe that, in general, computing $post^{\#}$ involves solving an NP-complete problem, and present two orthogonal solutions to this problem in Sect. 5 and 6, respectively. Each of them leads to a polynomial-time algorithm. The first solution works for a restricted class of systems and arbitrary abstractions, while the second restricts the class of abstractions that are used but can be applied to arbitrary systems. In Sect. 7 we present an abstraction-refinement scheme which allows to progressively refine the precision of the abstractions while keeping good control of the time required to compute the $(post^{\#})^*$ operator, i.e., the operator yielding the set of reachable abstract states. Section 8 reports on experimental results obtained from an implementation of the approaches of Sect. 5 and 6.

Related work. Locality-based abstractions have been used before in the literature, but to the best of our knowledge not with the generality presented here. A particular case of locality-based abstraction are the *Cartesian abstractions* of [2], in which a set of tuples is approximated by the smallest Cartesian product containing this set. It corresponds to the case in which we have an observer for each variable (i.e., the observer can only see this variable, and nothing else). Another particular case that has been independently rediscovered several times is the *pairs* abstraction, in which we have an observer for each (unordered) pair of variables. In [3,4,5], this abstraction is used to overapproximate the pairs $\{l, l'\}$ of program points of a concurrent program such that during execution the control can simultaneously be at l, l'. In [6], it is used to overapproximate the pairs of places of a Petri net that can be simultaneously marked, and the abstraction is proved to be exact for the subclass of T-nets, also called marked graphs. In Graphplan, an approach to the solution of propositional planning problems [7,8], it is used to overapproximate the set of states reachable after at most n steps.

Prerequisites. The reader is expected to be familiar with the abstract interpretation framework and with the manipulation of symbolic data structures based on deterministic automata such as binary decision diagrams [9].

Full version. A version of the paper containing all proofs is available at `http://www.ulb.ac.be/di/ssd/cfv/publications.html`.

2 Preliminaries

System model. We fix a finite set V of *values* (in our examples we use $V = \{0, 1\}$). A *state* is a function $s \colon X \to V$, where $X = \{x_1, \ldots, x_n\}$ is a set of *state variables*. We also represent a state s by the tuple $(s[1], \ldots, s[n])$, where $s[i]$ is an abbreviation for $s(x_i)$. The set of all states over the set X of variables is denoted by $\mathscr{S}$.

Let X' be a disjoint copy of X. A *transition* t is a subset of $\mathscr{S} \times \mathscr{S}$, which we represent as a predicate $t(X, X')$, i.e., $(s, s') \in t$ if and only if $t(s, s')$ is true.

A *system* is a pair $Sys = (X, T)$ where X is a finite set of variables and T is a finite set of transitions. We define the *transition relation* $R \subseteq \mathscr{S} \times \mathscr{S}$ as the union of all the transitions of T.

Given a set of states S, we define the *successors* of S, denoted by $post[Sys](S)$, as the set of states s' such that $R(s, s')$ for some $s \in S$, and the *predecessors* of S, denoted by $pre[Sys](S)$, as the set of states s' such that $R(s', s)$ for some $s \in S$. We also write $post(S)$ or $pre(S)$ if the system Sys is clear from the context. We use the following notations: $post^0(S) = S$, $post^{i+1}(S) = post(post^i(S))$ for every $i \geq 0$, and $post^*(S) = \bigcup_{n \in \mathbb{N}} post^n(S)$. We use analogous notations for *pre*. A state s is *reachable* from S if $s \in post^*(S)$.

Partial states. Let $V^+ = V \cup \{\mathbf{u}\}$ where $\mathbf{u}$, disjoint from V, is the undefined value. It is convenient to define a partial order $\succeq$ on V^+, given by

$$v \succeq v' \overset{\text{def}}{\iff} (v' = \mathbf{u} \vee v = v') \ .$$

A *partial state* is a function $p \colon X \to V^+$. The set of all partial states is denoted by $\mathscr{P}$. The *support* of a partial state p is the set of indices $i \in \{1, \ldots, n\}$ such that $p[i] \neq \mathbf{u}$. We extend the partial order $\succeq$ to partial states:

$$p \succeq p' \overset{\text{def}}{\iff} \bigwedge_{x \in X} \big(p(x) \succeq p'(x)\big)$$

and to sets of partial states:

$$P \succeq P' \overset{\text{def}}{\iff} \forall p \in P \, \exists p' \in P' \colon p \succeq p' \ .$$

Given a partial state p, we define its *upward and downward closure* as $p\!\uparrow = \{p' \in \mathscr{P} \mid p' \succeq p\}$ and $p\!\downarrow = \{p' \in \mathscr{P} \mid p \succeq p'\}$, respectively. We extend these two notions to sets of partial states in the natural way. We say that P is *upward* or *downward* closed if $P\!\uparrow = P$ or $P\!\downarrow = P$, respectively. We also say that P is a *uc-set* or a *dc-set.*

Finally, we also define $p\!\Uparrow = p\!\uparrow \cap \mathscr{S}$, and extend the notation to sets of states.

3 Locality-Based Abstractions

Fix a system Sys and a set I of initial states of Sys. We say that a partial state p is *reachable* from I if *some* state $s \succeq p$ is reachable from I. Observe that with this definition p is reachable if and only if all partial states in the downward closure $p\!\downarrow$ are reachable. So the pieces of information we have about reachability of partial states can be identified with downward closed subsets of $\mathscr{P}$.

Assume now that the only dc-sets we have access to are those included in some dc-set $D \subseteq \mathscr{P}$, called in the rest of the paper a *domain.* If a state s is reachable, then all the elements of $s\!\downarrow \cap D$ are reachable by definition. However, the contrary does not necessarily hold, since we may have $s \notin D$. In our abstractions we overapproximate by declaring s reachable if all the elements of $s\!\downarrow \cap D$

are reachable, i.e., if all the information we have access to is compatible with s being reachable.

Intuitively, we can look at D as the union of sets $D_1, \ldots, D_n$, where all the partial states in D_i have the same support, i.e., a partial state $p \in D_i$ satisfies $p[i] = \mathbf{u}$ only if all partial states $p' \in D_i$ satisfy $p'[i] = \mathbf{u}$. The sets D_i correspond to the pieces of information that the different observers have access to. Notice that we can have a domain D_i like, say $D_i = \{\langle 0, 0, \mathbf{u}\rangle, \langle 1, 0, \mathbf{u}\rangle\}{\downarrow}$ in which the observer is only allowed to see some local states of the first two components, but not others, like $\langle 1, 1, \mathbf{u}\rangle$.

Recall that the powerset lattice $PL(A)$ associated to a set A is the complete lattice having the powerset of A as carrier, and union and intersection as least upper bound and greatest lower bound operations, respectively. In our abstractions the concrete lattice is the powerset lattice $PL(\mathscr{S})$ of the set of states $\mathscr{S}$.

We fix a domain $D \subseteq \mathscr{P}$, and define the *downward powerset lattice* $DPL(D)$ associated to D as the restriction of $PL(D)$ to the dc-sets included in D. That is, the carrier of $DPL(D)$ is the set of dc-subsets of D (which, since D is downward closed, contains D itself), and the least upper bound and greatest lower bound operations are union and intersection. Notice that $DPL(D)$ is well-defined because the union and intersection of a family of dc-sets is a dc-set. The abstract lattice of a locality-based abstraction is $DPL(D)$, and the concretization and abstraction mappings are defined as follows:

$$\begin{aligned}
\alpha(S) &\stackrel{\text{def}}{=} S{\downarrow} \cap D & &\text{for any } S \in PL(\mathscr{S}) \\
\gamma(P) &\stackrel{\text{def}}{=} \{s \in \mathscr{S} \mid s{\downarrow} \cap D \subseteq P\} & &\text{for any } P \in DPL(D) \\
&= \mathscr{S} \setminus (D \setminus P){\Uparrow} .
\end{aligned}$$

Example 1. Consider the set of values V and the state variables X defined by $V = \{0, 1\}$ and $X = \{x_1, x_2, x_3\}$, respectively. The domain of pairs over X is given by

$$D_2 = \{(n, m, \mathbf{u}), (n, \mathbf{u}, m), (\mathbf{u}, n, m) \mid n, m \in \{0, 1\}\}{\downarrow} \quad .$$

For the set $S = \{\langle 1, 1, 0\rangle, \langle 1, 0, 0\rangle, \langle 0, 1, 0\rangle\}$ we get

$$\alpha(S) = \{\langle 1, 1, \mathbf{u}\rangle, \langle 1, 0, \mathbf{u}\rangle, \langle 0, 1, \mathbf{u}\rangle, \langle 1, \mathbf{u}, 0\rangle, \langle 0, \mathbf{u}, 0\rangle, \langle \mathbf{u}, 1, 0\rangle, \langle \mathbf{u}, 0, 0\rangle\}{\downarrow}$$

and $(\gamma \circ \alpha)(S) = S$, i.e., in this case no information is lost.

Consider now the domain

$$D_1 = \{(n, \mathbf{u}, \mathbf{u}), (\mathbf{u}, n, \mathbf{u}), (\mathbf{u}, \mathbf{u}, n), (\mathbf{u}, \mathbf{u}, \mathbf{u}) \mid n \in \{0, 1\}\} \quad .$$

In this case we get

$$\begin{aligned}
(\gamma \circ \alpha)(S) &= \gamma(\{\langle 1, \mathbf{u}, \mathbf{u}\rangle, \langle 0, \mathbf{u}, \mathbf{u}\rangle, \langle \mathbf{u}, 1, \mathbf{u}\rangle, \langle \mathbf{u}, 0, \mathbf{u}\rangle, \langle \mathbf{u}, \mathbf{u}, 0\rangle, \langle \mathbf{u}, \mathbf{u}, \mathbf{u}\rangle\}) \\
&= \{0, 1\} \times \{0, 1\} \times \{0\}
\end{aligned}$$

and in general $(\gamma \circ \alpha)(S)$ is the smallest cartesian product of subsets of V containing S, matching the cartesian abstractions of [2] [1].

Observe that for $D = \mathscr{P}$ we obtain

$$\alpha(S) = S{\downarrow} \qquad \text{for any } S \in PL(\mathscr{S})$$
$$\gamma(P) = P \cap \mathscr{S} \qquad \text{for any } P \in DPL(D)$$

and so $(\gamma \circ \alpha)(S) = S$, i.e., no information is lost.

The concrete $PL(\mathscr{S})$ and abstract $DPL(D)$ domains and the abstraction $\alpha\colon PL(\mathscr{S}) \mapsto DPL(D)$ and concretization $\gamma\colon DPL(D) \mapsto PL(\mathscr{S})$ maps form a *Galois connection*, denoted by $PL(\mathscr{S}) \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} DPL(D)$, for every domain D.

Proposition 1. *For every domain D, $PL(\mathscr{S}) \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} DPL(D)$.*

3.1 The $post^{\#}$ Operator

We define the function $post^{\#}[Sys, D]\colon DPL(D) \to DPL(D)$:

$$post^{\#}[Sys, D] \overset{\text{def}}{=} \lambda P.(\alpha \circ post[Sys] \circ \gamma)(P) \ .$$

We shorten $post^{\#}[Sys, D]$ to $post^{\#}$ if the system and the domain are clear from the context. We have the following characterization of $post^{\#}[Sys, D]$.

Proposition 2. *Let Sys and D be a system and a domain, respectively. For every $P \in DPL(D)$ and for every $p \in \mathscr{P}$*

$$p \in post^{\#}(P) \Longleftrightarrow p \in D \wedge \neg(pre(p{\Uparrow}) \subseteq (D \setminus P){\Uparrow}) \ .$$

Proof (of Proposition 2).

$$\begin{aligned} p \in post^{\#}(P) &\Leftrightarrow p \in (\alpha \circ post \circ \gamma)(P) \\ &\Leftrightarrow p \in D \wedge p \in ({\downarrow} \circ post \circ \gamma)(P) && \text{(Def. of } \alpha) \\ &\Leftrightarrow p \in D \wedge (p{\Uparrow} \cap (post \circ \gamma)(P) \neq \emptyset) \\ &\Leftrightarrow p \in D \wedge (pre(p{\Uparrow}) \cap \gamma(P) \neq \emptyset) \\ &\Leftrightarrow p \in D \wedge (pre(p{\Uparrow}) \cap \big(\mathscr{S} \setminus (D \setminus P){\Uparrow}\big) \neq \emptyset) && \text{(Def. of } \gamma) \\ &\Leftrightarrow p \in D \wedge \neg(pre(p{\Uparrow}) \subseteq (D \setminus P){\Uparrow}) \end{aligned}$$

□

Using standard results of abstract interpretation we get for every set of states S that $(post^{\#})^*$ is a sound abstraction of $post^*$, i.e.:

$$post^*(S) \subseteq (\gamma \circ (post^{\#})^* \circ \alpha)(S) \qquad \text{for every } S \in PL(\mathscr{S}).$$

[1] Actually, the functions α and γ of [2] are slightly different, but their composition is the same as here.

4 The Complexity of Computing *post*$^{\#}$

In the rest of the paper we assume that sets of (partial) states are symbolically represented as *multi-valued decision diagrams* (MDD) (see [10] for more details) over the set of variables X with a fixed variable order. The cardinality of X will be denoted $|X|$. Given a set P of partial states we denote the MDD representing P by $P^{\mathcal{M}}$ and the size of $P^{\mathcal{M}}$ by $|P^{\mathcal{M}}|$. We also assume that each transition t of a system $Sys = (X, T)$ is symbolically represented as a MDD $t^{\mathcal{M}}$ over variables X, X' with a fixed variable order whose projection onto X coincides with the fixed order on X. The *size* of Sys is defined as $\sum_{t \in T} |t^{\mathcal{M}}| + |X|$ and denoted by $|Sys|$.

We consider the following decision problem.

Definition 1. *The problem POST*$^{\#}$ *is defined as follows:*
Instance: *a system* $Sys = (X, T)$, *an element* $p \in D$ *and two MDDs* $D^{\mathcal{M}}$, $P^{\mathcal{M}}$, *where* D *is a non-empty domain and* $P \in DPL(D)$.
Question: $p \in post^{\#}[Sys, D](P)$ *?*

We say that a class of systems $\mathcal{C}$ is *polynomial* if the restriction $\text{POST}^{\#}_{\mathcal{C}}$ of $\text{POST}^{\#}$ to instances in which the system *Sys* belongs to $\mathcal{C}$ can be solved in polynomial time. Unfortunately, as we are going to show, unless P=NP holds, even very simple classes of systems are not polynomial. Before proceeding, we need a time complexity bound for some operation on MDD.

Proposition 3. *Let* $p \in \mathscr{P}$ *and* $S^{\mathcal{M}}$ *be a MDD for* $S \subseteq \mathscr{P}$. *We can decide in* $O(|X| + |S^{\mathcal{M}}|)$ *time if there exists* $s \in S$ *such that* $p \succeq s$.

Proof (of Proposition 3). We use a simple marking algorithm. Initially we mark the root node of $S^{\mathcal{M}}$. If a node m labelled by x_i is marked, we mark all successors n of m such that the edge $e = (n, m)$ is labelled with a $v_i \in V^+$ satisfying $p(x_i) \succeq v_i$. The state s exists iff at the end of the algorithm the accepting node is marked. □

The following proposition is proved by means of a simple reduction from the 3-colorability problem on graphs.

Proposition 4. *The following problem is NP-complete:*
Instance: *a set* X *of variables, and two MDDs* $D^{\mathcal{M}}$, $P^{\mathcal{M}}$, *where* D *is a non-empty domain on* X *and* $P \in DPL(D)$.
Question: $\gamma(P) \neq \emptyset$ *?*
In particular, if P≠NP then there is no polynomial time algorithm to compute $\gamma(P)^{\mathcal{M}}$.

We are now able to present the main result of this section. Fix $V = \{0, 1\}$ and let $\{Sys_n\}_{n \geq 1}$ be the family of systems given by $Sys_n = (X_n, \{t_n\})$, $X_n = \{x_1, \ldots x_n\}$ and $t_n = \mathscr{S} \times \mathscr{S}$. Intuitively Sys_n is a system with n state variables and a unique transition t_n such that for any pairs of states s, s' we find that $(s, s') \in t_n$.

Proposition 5. *If the class $\mathcal{C} = \{Sys_n\}_{n\geq 1}$ of systems is polynomial, then P=NP.*

Proof. We reduce the problem of Prop. 4 to $\text{POST}^{\#}_{\mathcal{C}}$. This shows that $\text{POST}^{\#}_{\mathcal{C}}$ is NP-complete and so if $\mathcal{C}$ is polynomial, then P=NP.

Given an instance X, $D^{\mathcal{M}}$, $P^{\mathcal{M}}$ of the problem of Prop. 4, we build in polynomial time the partial state $\mathbf{u}^{|X|}$ ($\mathbf{u}^{|X|} \in D$ for any $D \neq \emptyset$) and the MDD $t^{\mathcal{M}}_{|X|}$ such that $t_{|X|} = \mathscr{S} \times \mathscr{S}$. The operator $post^{\#}$ is given by $(\alpha \circ post \circ \gamma)$ and so we have $\gamma(P) \neq \emptyset$ iff $\mathbf{u}^{|X|} \in post^{\#}[Sys_{|X|}, D](P)$. □

This result shows that we do not have much hope of finding a broad, interesting class of polynomial systems. In the next sections we present two possible ways of dealing with this problem.

5 Partial Reachability

In this section we show that, if we change the concrete lattice in our abstractions by extending reachability also to partial states, then an interesting class of systems becomes polynomial. From now on, we assume the following ordering on $X = \{x_1, \ldots, x_n\}$ and its disjoint copy X': $x_1 < x'_1 < \cdots < x_n < x'_n$.

We define the notion of kernel of a transition. Intuitively, the kernel of a transition is the set of variables that are "involved" in it.

Definition 2. *Let $t(X, X')$ be a transition and let $Y \subseteq X$ be the smallest subset of X such that*

$$t(X, X') \equiv \hat{t}(Y, Y') \wedge \bigwedge_{x \in X \setminus Y} (x = x')$$

for some relation $\hat{t}$. We call $\hat{t}$ the kernel *of t, Y the* kernel variables *and $|Y|$ the* kernel width. *Given a partial state $p \in \mathscr{P}$, we denote by $\hat{p}$ the partial state given by*

$$\hat{p}[i] = \begin{cases} p[i] & \text{if } x_i \text{ belongs to the kernel variables of } t, \\ \mathbf{u} & \text{otherwise,} \end{cases}$$

and $\tilde{p}$ the partial state given by

$$\tilde{p}[i] = \begin{cases} p[i] & \text{if } x_i \text{ does not belong to the kernel variables of } t, \\ \mathbf{u} & \text{otherwise.} \end{cases}$$

We identify a partial state p and the pair $(\hat{p}, \tilde{p})$.

We need to extend the transitions to partial states.

Definition 3. *Let $Sys = (X, T)$ be a system and let $t \in T$. The* extended transition *$\underline{t} \subseteq \mathscr{P} \times \mathscr{P}$ is defined as follows:*

$$\underline{t}(p_1, p_2) \overset{\text{def}}{\iff} \exists p \in \mathscr{P} \colon \hat{t}(\hat{p}_1, p) \wedge p \succeq \hat{p}_2 \wedge \tilde{p}_1 = \tilde{p}_2 \ .$$

Given a system $Sys = (X, T)$, we define the extended transition relation *$\underline{R}$ of Sys as the union of all its extended transitions.*

The intuition behind this definition is as follows: If we know that p_1 is reachable (i.e., that some state $s \succeq p_1$ is reachable) and that $\underline{t}(p_1, p_2)$ holds, then we already have enough information to infer that p_2 is reachable. Let us see why. We know the values of all the variables involved in t (this is $\hat{p}_1$), and we know that we can reach $(p, \tilde{p}_1)$ from $(\hat{p}_1, \tilde{p}_1)$ (because $\hat{t}(\hat{p}_1, p)$). Now, since we can reach $(p, \tilde{p}_1)$ and we know that $p \succeq \hat{p}_2$ and $\tilde{p}_1 = \tilde{p}_2$, we can infer that p_2 is also reachable.

It is easy to show that the restriction of the extended reachability relation to states coincides with the reachability relation.

Lemma 1. *For any $t \in T$ and any $s_1, s_2 \in \mathscr{S}$, we have $t(s_1, s_2)$ iff $\underline{t}(s_1, s_2)$.*

Proof (of Lemma 1). Since s_2 is a state, $p \succeq \hat{s}_2$ holds if and only if $p = \hat{s}_2$, and so

$$\underline{t}(s_1, s_2) \Leftrightarrow (\hat{t}(\hat{s}_1, \hat{s}_2) \wedge \tilde{s}_1 = \tilde{s}_2) \Leftrightarrow t(s_1, s_2) \ . \qquad \square$$

In order to obtain a Galois connection, we extend the functions α, γ to $\underline{\alpha}\colon DPL(\mathscr{P}) \to DPL(D)$ and $\underline{\gamma}\colon DPL(D) \to DPL(\mathscr{P})$ in the obvious way:

$$\begin{aligned}
\forall P \in DPL(\mathscr{P})\colon \ \underline{\alpha}(P) &\stackrel{\text{def}}{=} P{\downarrow} \cap D = P \cap D \quad (P \text{ is a } \textit{dc-set}) \\
\forall P \in DPL(D)\colon \ \underline{\gamma}(P) &\stackrel{\text{def}}{=} \{p \mid p{\downarrow} \cap D \subseteq P\} \\
&= \mathscr{P} \setminus (D \setminus P){\uparrow} \ .
\end{aligned}$$

Proposition 6. *For every domain D, $DPL(\mathscr{P}) \underset{\underline{\gamma}}{\overset{\underline{\alpha}}{\rightleftharpoons}} DPL(D)$.*

5.1 The $\underline{post}^{\#}$ Operator

We extend *post* and *pre* to $\underline{post}$ and $\underline{pre}$ on partial states by declaring $p' \in \underline{post}(p)$ and $p \in \underline{pre}(p')$ iff $\underline{R}(p, p')$. We have the following useful property:

Lemma 2. *Fix an arbitrary system, for every $p \in \mathscr{P}$, $\underline{pre}(p{\uparrow}) = \underline{pre}(p){\uparrow}$ and $\underline{post}(p{\uparrow}) = \underline{post}(p){\uparrow}$.*

The set $\underline{post}^{\#}[Sys, D](P)$ is given by

$$\{p_2 \in D \mid \exists p_1 \colon \underline{R}(p_1, p_2) \wedge \neg\big(\exists p_3 \colon p_3 \in (D \setminus P) \wedge (p_1 \succeq p_3)\big)\} \ .$$

Notice that, given MDDs $D^{\mathcal{M}}$, $P^{\mathcal{M}}$, $\underline{R}^{\mathcal{M}}$ and $\succeq^{\mathcal{M}}$, the set $\underline{post}^{\#}[Sys, D](P)$ can be computed symbolically using classical operations provided by any MDD package.

The following result, which makes use of Lemmata 1 and 2, shows that the $post^{\#}$ operator is a better approximation to *post* than $\underline{post}^{\#}$, i.e., replacing $post^{\#}$ by $\underline{post}^{\#}$ leads to a loss of precision.

Proposition 7. *Fix a system and a domain D. For every $P \in DPL(D)$, $post^{\#}(P) \subseteq \underline{post}^{\#}(P)$, but the converse does not hold.*

Proof. The first part is an easy consequence of the definitions, and can be found in the full version of the paper. Here we provide a detailed example proving the non inclusion of $\underline{post}^{\#}(P)$ in $post^{\#}(P)$.

Fix $V = \{0, 1, 2\}$ and $Sys = (X, T)$ with $X = \{x_1, x_2, x_3, x_4\}$, $T = \{t_1, t_2, t_3, t_4\}$ and such that

$$
\begin{array}{lll}
t_1(X, X') \equiv \hat{t}_1(Y, Y') \wedge x_3 = x_3' & \hat{t}_1 = \{(\langle 0,0,0\rangle, \langle 1,1,1\rangle)\} & Y = X \setminus \{x_3\} \\
t_2(X, X') \equiv \hat{t}_2(Y, Y') \wedge x_2 = x_2' & \hat{t}_2 = \{(\langle 0,0,0\rangle, \langle 1,1,2\rangle)\} & Y = X \setminus \{x_2\} \\
t_3(X, X') \equiv \hat{t}_3(Y, Y') \wedge \begin{pmatrix} x_1 = x_1' \\ x_4 = x_4' \end{pmatrix} & \hat{t}_3 = \{(\langle 0,0\rangle, \langle 1,1\rangle)\} & Y = \{x_2, x_3\} \\
t_4(X, X') \equiv \hat{t}_4(Y, Y') \wedge x_4 = x_4' & \hat{t}_4 = \{(\langle 1,1,1\rangle, \langle 2,2,2\rangle)\} & Y = X \setminus \{x_4\}
\end{array}
$$

The domain D is the set of partial states $p \in \{0, 1, 2, \mathbf{u}\}^4$ such that for at most 2 indices i, j of $\{1, 2, 3, 4\}$: $p[i] \neq \mathbf{u}$ and $p[j] \neq \mathbf{u}$. The set of initial state I is given by $\{\langle 0, 0, 0, 0\rangle\}$. The set $(\underline{post}^{\#} \circ \underline{\alpha})(I{\downarrow})$, denoted F, is given by

$$
\begin{aligned}
F = \{&\langle 1,1,\mathbf{u},\mathbf{u}\rangle, \langle 1,\mathbf{u},\mathbf{u},1\rangle, \langle \mathbf{u},1,\mathbf{u},1\rangle, \langle 1,\mathbf{u},0,\mathbf{u}\rangle, \langle \mathbf{u},1,0,\mathbf{u}\rangle, \langle \mathbf{u},\mathbf{u},0,1\rangle \\
&\langle 1,\mathbf{u},1,\mathbf{u}\rangle, \langle 1,\mathbf{u},\mathbf{u},2\rangle, \langle \mathbf{u},\mathbf{u},1,2\rangle, \langle 1,0,\mathbf{u},\mathbf{u}\rangle, \langle \mathbf{u},0,1,\mathbf{u}\rangle, \langle \mathbf{u},0,\mathbf{u},2\rangle, \\
&\langle \mathbf{u},1,1,\mathbf{u}\rangle, \langle 0,1,\mathbf{u},\mathbf{u}\rangle, \langle \mathbf{u},1,\mathbf{u},0\rangle, \langle 0,\mathbf{u},1,\mathbf{u}\rangle, \langle \mathbf{u},\mathbf{u},1,0\rangle, \langle 0,\mathbf{u},\mathbf{u},0\rangle\}{\downarrow}\ .
\end{aligned}
$$

It is routine to check that $(post^{\#} \circ \alpha)(I)$ and $(\underline{post}^{\#} \circ \underline{\alpha})(I{\downarrow})$ coincide. Observe that $\langle 1, 1, 1, \mathbf{u}\rangle \in \underline{\gamma}(F)$ but

$$\{\langle 1,1,1,0\rangle, \langle 1,1,1,1\rangle, \langle 1,1,1,2\rangle\} \cap \underline{\gamma}(F) = \emptyset\ .$$

Now consider the second iteration. In this case we find that $\langle 2, 2, \mathbf{u}, \mathbf{u}\rangle \in \underline{post}^{\#}(F)$ but $\langle 2, 2, \mathbf{u}, \mathbf{u}\rangle \notin post^{\#}(F)$ which proves our claim. □

The loss of precision of $\underline{post}^{\#}$ is compensated by its better properties. We have the following characterization of $\underline{post}^{\#}[Sys, D](P)$.

Proposition 8. *Let Sys and D be a system and a domain, respectively. For every $P \in DPL(D)$, for every $p \in \mathscr{P}$*

$$p \in \underline{post}^{\#}(P) \Longleftrightarrow p \in D \wedge \neg(\underline{pre}(p) \succeq (D \setminus P))\ .$$

Proof (of Proposition 8).

$$
\begin{aligned}
p \in \underline{post}^{\#}(P) &\Leftrightarrow p \in (\underline{\alpha} \circ \underline{post} \circ \underline{\gamma})(P) \\
&\Leftrightarrow p \in D \wedge p \in (\underline{post} \circ \underline{\gamma})(P) && \text{(Def. of } \underline{\alpha}) \\
&\Leftrightarrow p \in D \wedge (\underline{pre}(p) \cap \underline{\gamma}(P) \neq \emptyset) \\
&\Leftrightarrow p \in D \wedge (\underline{pre}(p) \cap (\mathscr{P} \setminus (D \setminus P){\uparrow}) \neq \emptyset) && \text{(Def. of } \underline{\gamma}) \\
&\Leftrightarrow p \in D \wedge \neg(\underline{pre}(p) \subseteq (D \setminus P){\uparrow}) \\
&\Leftrightarrow p \in D \wedge \neg(\underline{pre}(p) \succeq (D \setminus P))
\end{aligned}
$$

□

This proposition shows the difference between computing $post^{\#}$ and $\underline{post}^{\#}$: In the first case we have to deal with $(D \setminus P)\Uparrow$, which can have a much more complex symbolic representation than $(D \setminus P)$. In the case of $\underline{post}^{\#}$ we only need to deal with the set $(D \setminus P)$ itself.

5.2 The Complexity of Computing $\underline{post}^{\#}$

Given a system Sys, we define the problem $\underline{\text{POST}}^{\#}$ as $\text{POST}^{\#}$, just replacing $post^{\#}$ by $\underline{post}^{\#}$. As seen in Prop. 8, we can decide $\underline{\text{POST}}^{\#}$ by checking whether $\underline{pre}(p) \succeq \overline{(D \setminus P)}$ holds. Consider the class of systems satisfying the following two conditions for every partial state p,

(a) $|\underline{pre}(p)|$ is bounded by a polynomial in $|X|$, and
(b) $\underline{pre}(p)^{\mathcal{M}}$ can be computed in polynomial time in $|X|$.

By Prop. 3, for $p' \in \underline{pre}(p)$, we can decide $\{p'\} \succeq (D \setminus P)$ in time $O(|D^{\mathcal{M}}| \cdot |P^{\mathcal{M}}| + |X|)$ and thus, given $\underline{pre}(p)^{\mathcal{M}}$, $D^{\mathcal{M}}$, and $P^{\mathcal{M}}$, we can decide $\underline{pre}(p) \succeq (D \setminus P)$ in polynomial time. Since $|\underline{pre}(p)^{\mathcal{M}}|$ is polynomial in $|X|$ and $|X|$ is $O(|Sys|)$, we can decide $\underline{\text{POST}}^{\#}$ in polynomial time. It follows that these systems are polynomial for $\underline{\text{POST}}^{\#}$.

We now show that an interesting class of systems satisfies (a) and (b). Intuitively, we look at a system on a set X as a set having $|X|$ components. Each variable describes the local state of the corresponding component.

Definition 4. *A system $Sys = (X, T)$ is* k-bounded *if the width of the kernel of all transitions of T is bounded by k.*

Loosely speaking, a system is k-bounded if its transitions involve at most k components. Many systems are k-bounded. For instance, consider systems communicating by point to point channels. If we describe the local state of a component/channel by one variable, then usually we have $k = 2$, because a transition depends on the current state of the receiving/sending component and on the state of the channel. Another example are token ring protocols, where each component communicates only with its left and right neighbours. These systems are at most 3-bounded.

Observe that each k-bounded system is equivalent to another one satisfying $|T| \leq |X|^k$: if there is $\hat{t}_i(Y_i, Y_i'), \hat{t}_j(Y_j, Y_j')$ such that $i \neq j$ but $Y_i = Y_j$, then we can replace $\hat{t}_i$ and $\hat{t}_j$ by $(\hat{t}_i \vee \hat{t}_j)(Y_i, Y_i')$.

Proposition 9. *Let p be a partial state of a k-bounded system. The set $\underline{pre}(p)$ contains at most $|X|^k \cdot |V^+|^k$ elements, and $\underline{pre}(p)^{\mathcal{M}}$ can be computed in time polynomial in $|X|$.*

Corollary 1. *For a fixed $k \geq 0$, the class of k-bounded systems is polynomial.*

6 Neighbourhood Domains

The polynomiality result of the last section is obtained at a price: we had to consider less precise abstractions, and we had to restrict ourselves to k-bounded systems. In this section we define an approach, applicable to arbitrary systems, that uses a class of domains called *neighbourhood domains*. Intuitively, in a neighbourhood domain the variables an observer has access to must be *neighbours* with respect to the order used to construct the MDDs. E.g., an observer may observe variables x_3, x_4, x_5, but not x_1, x_8.

We say that a class $\mathcal{D}$ of domains is *polynomial* if the restriction $\mathrm{POST}^{\#}_{\mathcal{D}}$ of $\mathrm{POST}^{\#}$ to instances in which the domain D belongs to $\mathcal{D}$ can be solved in polynomial time.

By Prop. 4 we know that, unless P=NP, there is no polynomial algorithm to compute $\gamma(\cdot)$. We define hereafter a class of domains which avoids this problem, i.e., for every set P in the domain, the $\gamma(P)^{\mathcal{M}}$ can be computed in time polynomial in $|X|$, $|P^{\mathcal{M}}|$ and $|D^{\mathcal{M}}|$.

Definition 5. *Let $x_1 < \cdots < x_n$ be a variable ordering for X and let $1 \leq k \leq |X|$. The k-*neighbourhood *domain D is defined as follows*

$$D(X) \equiv \bigvee_{V_i \in \mathcal{V}} \Big(\bigwedge_{x \in X \setminus V_i} (x = \mathbf{u}) \Big)$$

where $\mathcal{V}$ is the set of all the sets of k consecutive variables, e.g., for $n \geq k+2$ we find that $\{x_2, \ldots, x_{2+k}\} \in \mathcal{V}$.

In what follows, we sometimes abbreviate $\bigwedge_{x \in X \setminus V_i} (x = \mathbf{u})$ to $D_i(X)$.

The following two propositions introduce the two key properties of neighbourhood domains:

Proposition 10. *Let D be a k-neighbourhood domain D, and let $P \in DPL(D)$. The MDD for the set $(D \setminus P)\!\uparrow^{\mathcal{M}}$ can be be computed in polynomial time in $|(D \setminus P)^{\mathcal{M}}|$ (and so, in particular, it is only polynomially larger than $(D \setminus P)^{\mathcal{M}}$).*

We prove a similar result for the computation of the downward closure.

Proposition 11. *Given a k-neighbourhood domain D and a set $S \subseteq \mathscr{S}$, the MDD for $(S\!\downarrow \cap D_i)$ with $V_i \in \mathcal{V}$ can be computed in polynomial time in $|D_i^{\mathcal{M}}| \cdot |S^{\mathcal{M}}|$.*

It follows that, for neighbourhood domains, both α and γ can be computed in polynomial time in their input size.

Proposition 12. *For a fixed $k \geq 0$, the class of k-neighbourhood domains is polynomial.*

Proof. Consider an instance of $\mathrm{POST}^{\#}$ in which D is a k-neighbourhood domain. We give a polynomial algorithm to decide if $p \in \mathit{post}^{\#}[\mathit{Sys}, D](P)$. By the definition of $\mathit{post}^{\#}$ and α, we have $p \in \mathit{post}^{\#}[\mathit{Sys}, D](P)$ iff there is a transition

t and states s, s' such that $s \in \gamma(P) \wedge t(s, s') \wedge s' \in p\Uparrow$. By Prop. 10, $\gamma(P)^{\mathcal{M}}$ over variables X can be computed in polynomial time in $|D^{\mathcal{M}}|$ and $|P^{\mathcal{M}}|$, and an MDD $p \Uparrow^{\mathcal{M}}$, over variables X' can be computed in polynomial time in $|X|$. The algorithm constructs, for each transition t of the system, an MDD for the formula $\gamma(P)^{\mathcal{M}} \wedge t(X, X') \wedge p\Uparrow^{\mathcal{M}} (X')$ and checks if it encodes the empty set. Since the construction and the check can be carried out in polynomial time, we are done. □

Moreover, while in the concrete system the number of image computations may also be exponential, here we get a much better bound. Given a k-neighbourhood domain, each of the $(|X| - (k-1))$ formulæ $D_i(X)$ has exactly $|V|^k$ satisfying partial states. This leads us to the following fact: for any k-neighbourhood domain, for any system and for any set I of initial states, the number of iterations required to reach the fixed point in the computation of $(post^{\#})^*(I)$ is bounded by $(|X| - (k-1)) \times |V|^k$. Choosing the domain adequately, we thus have a way to control the complexity of computing $(post^{\#})^*(I)$. In practice this suggests the following strategy: if the *post* image computation is costly we can reduce the number of iterations needed to reach the fixed point by choosing a k-neighbourhood domain with $k << |X|$, of course at the prize of losing precision.

7 Abstraction Refinement

In this section, we describe an abstraction-refinement loop for testing reachability using the partial-reachability method. Given a system $Sys = (X, T)$, a set of initial states I, and a partial state u. Our goal is to check whether u is reachable, i.e. whether $u\Uparrow \cap post^*(I) \neq \emptyset$.

Our method starts from a (given) initial domain D and computes the reachable states in the abstraction, i.e. $(\underline{post}^{\#})^*(I\downarrow)$. If the latter includes u, we check if the imprecision caused by choosing the domain D might be responsible for the positive result. If so, we refine D accordingly.

More precisely, our scheme consists of the following two steps:

Search. Compute the sequence $F_0 = \underline{\alpha}(I\downarrow)$, $In_0 = Out_0 = \emptyset$, and then for $i \geq 0$:

$$\begin{aligned} T_{i+1} &= \{\, (p, p') \mid p \in \underline{\gamma}(F_i),\ p' \in (\underline{\alpha} \circ \underline{post})(p) \,\} \\ F_{i+1} &= F_i \cup \{\, p' \mid \exists p \colon (p, p') \in T_{i+1} \,\} \\ In_{i+1} &= In_i \cup \{\, (p, p') \in T_{i+1} \mid p \in D \,\} \\ Out_{i+1} &= Out_i \cup \{\, (p, p') \in T_{i+1} \mid p \in \mathscr{P} \setminus D \,\}. \end{aligned}$$

Stop when the sequence of F_is reaches a fixed point. We denote the values of the sets in the fixed point as F_f, In_f, Out_f, respectively.

Notice that T_i records a reachability relation between partial states, where the left components can be either previously computed partial states in D or partial states whose reachability was (potentially wrongly) 'inferred' by the concretization. We then have $F_{i+1} = F_i \cup \underline{post}^{\#}(F_i)$. The sequence of In sets records the

reachability relation between states for which no inference was used, whereas *Out* records the relations for which inference was used, i.e. the places where potential imprecision was introduced.

By Prop. 8, the T_i sets can be computed efficiently.

Refine. If $u \notin \underline{\gamma}(F_f)$, then by Prop. 7 u is unreachable, and we stop with a negative result.

Otherwise, if $u \notin D$, then we inferred reachability of u from the reachability of several partial states. We then refine D to $D \cup u\!\downarrow$, which forces the next iteration to 'watch' the partial state u explicitly. (Notice that we could have done this before the first iteration, but then again we might be able to refute reachability of u in the first iteration without doing this.)

Failing both tests, we check whether there is a real trace from an initial state to u. For this, we compute backwards reachability using the relation In_f. We conclude that u is reachable in the concrete system if $\exists i \in \underline{\alpha}(I\!\downarrow)\colon (i,u) \in In_f^*$. Executing this check step for step also gives us the ability to output a witness path for u's reachability.

Otherwise, u was reachable in the abstraction because of a step contained in Out_f. To prove concrete reachability of u, we must prove that the partial source states of these steps were indeed reachable. Thus, we compute the set $A = \{\, p \mid In_f^*(p,u) \,\}$ and then refine D to $D \cup \{\, p \mid \exists p' \in A\colon (p,p') \in Out_f \}$.

Our approach is different from the usual CEGAR approach (see [11] for more details), where one tests whether an abstract counterexample found in the search phase is spurious. If it is, one refines the abstraction to prevent that counterexample from being found again. In our approach, we cannot tell whether a counterexample is spurious or not; we can merely test whether potentially imprecise information was used. If the counterexample was spurious, our refinement prevents it from being found again. If the counterexample was real, then our refinement gathers additional information to prove the counterexample correct.

Extensive work (see [12,13,14]) investigates the connection between abstract model checking, and in particular the CEGAR approach, and the domain refinement used in abstract interpretation. As a future work we plan to investigate the connection but relatively to the refinement technique we proposed in this section.

8 Experiments

We have produced a prototype implementation of the approaches of Sect. 5 (with abstraction refinement) and 6 (without abstraction refinement), and applied it to two well-known examples. The examples only use boolean variables, and so we use BDDs instead of MDDs. Since our implementation is preliminary and our main motivation is to provide a space-efficient method, we only report on the sizes of the BDDs used to decide a property. We compare them with the BDD size of the full set of reachable states, which is computed using NuSMV [15].

8.1 Dining Philosophers Example

Our first example is a deterministic non-symmetric solution to the dining philosophers problem taken from [16]. The model uses two arrays, one for the forks and the other for the philosophers, both of size N, the number of philosophers. Each fork is represented by two bits, and each philosopher by three. For our experiments, we use two different 'natural' variable orders.

(A) The first order puts the bits for the forks at the top and the philosophers at the bottom, while each array element is stored with its most significant bit at the top.
(B) The second order interleaves forks and philosophers, i.e. we put the first fork at the top, then the first philosopher, then the second fork etc.

The sizes of the BDDs encoding the full set of reachable states are listed (for orders A and B and various values of N) in the left half of Table 1. As can be seen, they strongly depend on the variable ordering, with order B working far better.

We consider the following three properties:

1. Is it possible that two neighbouring philosophers eat at the same time? (This property is false in the model.)
2. Is it possible for all forks to be taken at the same time? (This property is true in the model.)
3. Is it possible for philosophers 1 and 3 to eat at the same time? (This property is true for all $N > 3$.)

Notice that, without a refinement loop, an abstraction can only prove that a set of states is *not* reachable, and so it can only be used to decide property 1. Since we have not implemented the refinement loop for the neighbourhood approach (see Sect. 6), we only apply it to this property. The partial-reachability approach is applied to all three properties.

In the neighbourhood approach, we can decide property 1 by taking $N+1$ and $k = 3$ for ordering A and B, respectively. The BDD sizes for $(post^{\#})^*(I)$ are shown in the right half of Table 1. We observe that, as for full reachability, the BDDs grow exponentially for ordering A and only linearly for ordering B.

Table 1. BDD sizes in Dining Philosophers example, part 1

	full reachability		neighb. approach	
N	ord. A	ord. B	ord. A	ord. B
2	26	25	13	18
3	64	41	40	36
4	140	57	82	54
7	1,204	105	304	108
10	9,716	153	670	162
15	311,284	273	1,600	252

Table 2. Results for Dining Philosophers, partial-reachability approach

	starting with 1 component						starting with 2 components					
	prop. 1		prop. 2		prop. 3		prop. 1		prop. 2		prop. 3	
N	$\|post^*\|$	$\#ref$	$\|post^*\|$	$\#ref$	$\|post^*\|$	$\#ref$	$\|post^*\|$	$\#ref$	$\|post^*\|$	$\#ref$	$\|post^*\|$	$\#ref$
2	52	5	34	3	n/a	n/a	38	0	46	2	n/a	n/a
3	128	4	112	4	109	7	73	0	144	4	73	0
4	217	4	321	5	89	5	112	0	426	5	152	4
7	523	4	6,781	8	167	5	259	0	6,300	8	335	4
10	919	4	??	?	245	5	451	0	??	?	563	4
15	1,807	4	??	?	375	5	871	0	??	?	1,043	4

However, the constant of the growth for ordering A is much smaller, i.e., the approach is far less sensitive to the variable order.

The results for the partial reachability approach are detailed in Table 2. We considered two different initial abstractions for the refinement loop. In the first one, we take one observer for each component (philosopher or fork); in the second, one observer for each pair of components (left and right part of Table 2, resp.). The $\#ref$ columns denote the number of refinements that were necessary to prove or disprove the properties. The column marked $|post^*|$ gives the number of BDD nodes used to represent $(\underline{post^{\#}})^*(I\!\downarrow)$ in the last refinement, where this number was highest. The representation of D was either nearly of the same size or significantly lower. The data for the orderings A and B are almost identical, and so only those for ordering A are shown. For properties 1 and 3, we observe the same pattern as in the neighbourhood case: the approach works well and is far less sensitive to the variable ordering. Looking closely, we observe that the 2-component initialization works better for property 1, presumably because the property is a conjunction of sub-properties concerning pairs of philosophers. For property 3, the 1-component initialization works better, probably because it concerns only 2 specific components. Property 2 is a case in which the locality-based approach works far worse than full reachability: The property is universally quantified, forcing the abstraction refinement to consider tuples ranging over all components.

8.2 Production Cell Example

Our second example is a model of the well-known production cell case study taken from [17]. Our encoding of the model has 15 variables with 39 bits altogether. We tested all fifteen safety properties mentioned in [17], but present the results for a few representative ones (the rest yielded similar results). The results are shown in Table 3.

Table 3 lists results for instantiations of the model with one and five plates. The number $|reach|$ is the BDD size of the reachable state space as computed by NuSMV, while $|post^*|$ and $\#ref$ have the same meanings as in Table 2.

The results show that while the reachable state space grows (linearly) with the number of plates, the partial-reachability approach is largely unaffected by their number. Moreover, while the number of refinement iterations varies (the

Table 3. Results for production cell example

	One plate $\|reach\| = 230$		Five plates $\|reach\| = 632$	
Prop	$\|post^*\|$	$\#ref$	$\|post^*\|$	$\#ref$
1	83	2	83	2
2	88	4	92	6
4	76	1	76	1
6	105	5	120	8
11	146	3	146	3

largest number of refinements was 13), the BDD sizes vary only by about 50% between the smallest and the largest example. As the number of plates increases, the space savings of the locality-based approach become significant.

In the neighbourhood approach, 4 out of the 15 properties could be proved with a neighbourhood domain of size $k = 2$. Independently of the number of plates, the number of BDD nodes representing the reachable state space was 129. A domain of size $k = 3$ was sufficient to verify another 7 properties; the number of BDD nodes increased to 208. The remaining properties could only be verified using full reachability, i.e. the neighbourhood approach did not have any advantage in this case.

9 Conclusions

We have presented locality-based abstractions, in which a state of the system is abstracted to the collection of views that some observers have of the state. Each observer has only access to some variables of the system. As pointed out in the introduction, special cases of locality-abstractions have been used in different contexts (planning, analysis of concurrent programs, concurrency theory). In this paper we have (1) generalized the abstractions used in other papers, (2) put them in the framework of abstract interpretation, (3) pointed out the bad complexity of the computation of the abstract successor operator for arbitrary locality-based abstractions, (4) provided two efficient solutions to this problem, and (5) evaluated these solutions on a number of examples. Our conclusion is that locality-based abstractions are a useful tool for the analysis of concurrent systems.

In our approach we have assumed that variables have a finite domain, and that if an observer has access to a variable, then it gets full information about its value. Both assumptions can be relaxed. For instance, locality-based abstractions can be easily combined with any of the usual abstractions on integer variables. It must only be required that clustered variables must be observable by the same observer.

References

1. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. POPL, ACM Press (1977) 238–252
2. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. In: Proc. TACAS. (2001) 268–283
3. Naumovich, G., Avrunin, G.S.: A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In: Proc. FSE. Volume 23, 6 of Software Engineering Notes., ACM Press (1998) 24–34
4. Naumovich, G., Avrunin, G.S., Clarke, L.A.: An efficient algorithm for computing *mhp* information for concurrent Java programs. In: Proc. FSE. Volume 1687 of LNCS. (1999) 338–354
5. Naumovich, G., Avrunin, G.S., Clarke, L.A.: Data flow analysis for checking properties of concurrent Java programs. In: Proc. ICSE, ACM Press (1999) 399–410
6. Kovalyov, A.: Concurrency relations and the safety problem for petri nets. In: Proc. ATPN. Volume 616 of LNCS. (1992) 299–309
7. Blum, A.L., Furst, M.L.: Fast planning through planning graph analysis. Artificial Intelligence **90** (1997) 279–298
8. Blum, A.L., Furst, M.L.: Fast planning through planning graph analysis. In: Proc. IJCAI. (1995) 1636–1642
9. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35** (1986) 677–691
10. Srinivasan, A., Kam, T., Malik, S., Brayton, R.K.: Algorithms for discrete function manipulation. In: IEEE/ACM ICCAD. (1990) 92–95
11. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50** (2003) 752–794
12. Ranzato, F., Tapparo, F.: Making abstract model checking strongly preserving. In: Proc. SAS. Volume 2477 of LNCS. (2002) 411–427
13. Giacobazzi, R., Quintarelli, E.: Incompleteness, counterexamples, and refinements in abstract model-checking. In: Proc. SAS. Volume 2126 of LNCS. (2001) 356–373
14. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. J. ACM **47** (2000) 361–416
15. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV version 2: An opensource tool for symbolic model checking. In: Proc. CAV. Volume 2404 of LNCS. (2002)
16. Zuck, L.D., Pnueli, A., Kesten, Y.: Automatic verification of probabilistic free choice. In: Proc. VMCAI. Volume 2294 of LNCS. (2002) 208–224
17. Heiner, M., Deussen, P.: Petri net based qualitative analysis - a case study. Technical Report I-08/1995, Brandenburg Tech. Univ., Cottbus (1995)

Type-Safe Optimisation of Plugin Architectures

Neal Glew[1], Jens Palsberg[2], and Christian Grothoff[3]

[1] Intel Corporation, Santa Clara, CA 95054, USA
aglew@acm.org
[2] UCLA Computer Science Dept, Los Angeles, CA 90095, USA
palsberg@ucla.edu
[3] Purdue University, Dept. of Computer Science, West Lafayette, IN 47907, USA
christian@grothoff.org

Abstract. Programmers increasingly implement plugin architectures in type-safe object-oriented languages such as Java. A virtual machine can dynamically load class files containing plugins, and a JIT compiler can do optimisations such as method inlining. Until now, the best known approach to type-safe method inlining in the presence of dynamic class loading is based on Class Hierarchy Analysis. Flow analyses that are more powerful than Class Hierarchy Analysis lead to more inlining but are more time consuming and not known to be type safe. In this paper we present and justify a new approach to type-safe method inlining in the presence of dynamic class loading. First we present experimental results that show that there are major advantages to analysing all locally available plugins at start-up time. If we analyse the locally available plugins at start-up time, then flow analysis is only needed at start-up time and when downloading plugins from the Internet, that is, when long pauses are expected anyway. Second, inspired by the experimental results, we design a new framework for type-safe method inlining which is based on a new type system and an existing flow analysis. In the new type system, a type is a pair of Java types, one from the original program and one that reflects the flow analysis. We prove that method inlining preserves typability, and the experimental results show that the new approach inlines considerably more call sites than Class Hierarchy Analysis.

1 Introduction

In a rapidly changing world, software has a better chance of success when it is extensible. Rather than having a fixed set of features, extensible software allows new features to be added on the fly. For example, modern browsers such as Firefox, Konqueror, Mozilla, and Viola [25] allow downloading of plug-ins that enable the browser to display new types of content. Using plugins can also help keep the core of the software smaller and make large projects more manageable thanks to the resulting modularisation. Plugin architectures have become a common approach to achieving extensibility and include well-known software such as Eclipse and Jedit.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 135–154, 2005.
© Springer-Verlag Berlin Heidelberg 2005

While good news for users, plug-ins architectures are challenging for optimising compilers. This paper investigates the optimisation of software that has a plug-in architecture and that is written in a type-safe object-oriented language. Our focus is on method inlining, one of the most important and most studied optimisations for object-oriented languages.

Consider the following typical snippet of Java code for loading and running a plugin.

```
String className = ...;
Class c = Class.forName(className);
Object o = c.newInstance();
Runnable p = (Runnable) o;
p.run();
```

The first line gets from somewhere the name of a plugin class. The list of plugins is typically supplied in the system configuration and loaded using I/O, preventing the compiler from doing a data-flow analysis to determine all possible plugins. The second line loads a plugin class with the given name. The third line creates an instance of the plugin class, which is subsequently cast to an interface and used.

In the presence of this dynamic loading, a compiler has two choices: either treat dynamic-loading points very conservatively or make speculative optimisations based on currently loaded classes only. The former can pollute the analysis of much of the program, potentially leading to little optimisation. The latter can potentially lead to more optimisation, but dynamically-loaded code might *invalidate* earlier optimisation decisions, and thus require the compiler to undo the optimisations. When a method inlining is invalidated by class loading, the run-time must *revirtualise* the call, that is, replace the inlined code with a virtual call. The observation that invalidations can happen easily in a system that uses plugins leads to the question:

> **Question:** If an optimising compiler for a plug-in architecture inlines aggressively, will it have to revirtualise frequently?

This paper presents experimental results for Eclipse and Jedit that quantify the potential invalidations and suggest how to significantly decrease the number of invalidations. We count which sites are likely candidates for future invalidation, which sites are unlikely to require invalidation, and which sites are guaranteed to stay inlined forever. These numbers suggest that speculative optimisation is beneficial and that invalidation can be kept manageable.

In addition to the goal of inlining more and revirtualising less, we want method inlining to preserve typability. This paper shows how to do inlining and revirtualisation in a way that preserves typability of the intermediate representation. The quest for preserving typability stems from the success of several compilers that use typed intermediate languages [9,15,16,17,26] to give debugging and optimisation benefits [16,24]. A bug in a compiler that discards type information might result in a run-time error, such as a segmentation violation,

that should be impossible in a typed language. On the other hand, if optimisations are type preserving, bugs can be found automatically by verifying that the compiler generates an intermediate respresentation that type checks. Additionally, preserving the types in the intermediate code may help guide other optimisations. So it is desirable to write optimisations so that they preserve typability.

Most of the compilers that use typed intermediate languages are "ahead-of-time" compilers. Similar benefits are desired for "just-in-time" (JIT) compilers. A step towards that goal was taken by the Jikes Research Virtual Machine [1] for Java, whose JIT compilers preserve and exploit Java's static types in the intermediate representations, chiefly for optimisation purposes. However, those intermediate representations are not typed in the usual sense—there is no type checker that guarantees type soundness (David Grove, personal communication, 2004). In two previous papers we presented algorithms for type-safe method inlining. The first paper [11] handles a setting *without* dynamic class loading, and the second paper [10] handles a setting *with* dynamic class loading, but with the least-precise flow analysis possible (CHA). In this paper we improve significantly on the second paper by presenting a new transformation and type system that together can handle a similar class of flow analyses as in the first paper.

Our Results. We make two contributions. Our first contribution is to present experimental numbers for inlining and invalidation. These numbers show that if a compiler analyses all plugins that are locally available, then dynamically loading from these plugins will lead to a miniscule number of invalidations. In contrast, when dynamically loading an unanalysed plugin, the run-time will have to consider a significantly larger number of invalidations. In order to ensure that loading unanalzed plugins happens less frequently, the compiler should analyse all of the local plugins using the most powerful technique available. That observation motivates our second contribution, which is a new framework for type-safe method inlining. The new framework handles dynamic class loading and a wide range of flow analyses. The main technical innovation is a technique for changing type annotations both at speculative devirtualisation time and at revirtualisation time, solving the key issue that we identified but side stepped in our previous paper [10]. As in both our previous papers, we prove a formalisation of the optimisation correct and type preserving. Using the most-precise flow analysis in the permitted class, our new framework achieves precision comparable to 0-CFA [18,21].

2 An Experiment

Using the plugin architectures Eclipse and Jedit as our benchmark, we have conducted an experiment that addresses the following questions:

- How many call sites can be inlined?
- How many inlinings remain valid and how many can be invalidated?

– How much can be gained by preanalysing the plugins that are statically available?

Preanalysing plugins can be beneficial. Consider the code in Figure 1. The analysis can see that the plugin calls method `m` in `Main` and passes it an `Main.B2`; since `main` also calls `m` with a `Main.B1`, it is probably not a good idea to inline the `a.n()` call in `m` as it will be invalidated by loading the plugin. The analysis can also see which methods are overridden by the plugin, in this case only `run` of `Runnable` is. The analysis must still be conservative in some places, for example at the instantiation inside of the `for` loop, as this statement could load any plugin. But the analysis can gather much more information about the program and make decisions based on likely invalidations by dynamically loading the known plugins.

Being able to apply the inlining optimisation in the first place still depends on the flow analysis being powerful enough to establish the unique target. Thus, the answer to each of the three questions depends on the static analysis that is used to determine which call sites have a unique target. We have experimented with four different interprocedural flow analyses, all implemented for Java bytecode, here listed in order of increasing precision (the first three support type preservation, the last one does not):

– Class Hierarchy Analysis (CHA, [7,8])
– Rapid Type Analysis (RTA, [2,3])
– subset-based, context-insensitive, flow-insensitive flow analysis for type-preserving method inlining (TSMI, [11]) and
– subset-based, context-insensitive, flow-insensitive flow analysis (0-CFA, [18,21]).

In order to show that deoptimisation is a necessity for optimising compilers for plugin architectures, we also give the results for a simple intraprocedural flow analysis ("local") which corresponds to the number of inlinings that will never have to be deoptimised, even if arbitrary new code is added to the system. The "local" analysis essentially makes conservative assumptions about all arguments, including the possibility of being passed new types that are not known to the analysis. A run-time system that cannot perform deoptimisation is limited to the optimisations found by "local" if loading arbitrary plugins is to be allowed.

The implementations of the five analyses share as much code as possible; our goal was to create the fairest comparison, not to optimise the analysis time. All of our experiments were run with at most 1.8 GB of memory. (1.8 GB is the maximum total process memory for the Hotspot Java Virtual Machine running on OS X as reported by `top` and also the memory limit specified at the command line using the `-Xmx` option.)

We use two benchmarks in our experiments:

Jedit 4.2pre13. A free programmer's text editor which can be extended with plugins from `http://jedit.org/`, 865 classes; analysed with GNU classpath 0.09, from `http://www.classpath.org`, 2706 classes.

```
class Main {
  static Main main;
  public static void main(String[] args) throws Exception {
    main = new Main();
    for (int i=0;i<args.length;i++) {
      Class c = Class.forName(args[i]);
      Runnable p = (Runnable) c.newInstance();
      p.run(); // virtual if loaded plugins define multiple run methods
    }
    main.m(new B1()); // can stay optimised for given Plugin
  }
  void m(A a) { a.n(); // needs to be virtual for given Plugin }
  static abstract class A {
    abstract void n();
  }
  static class B1 extends A {
    void n() { }
  }
  static class B2 extends Main.A {
     void n() { }
  }
}
class Plugin implements Runnable {
  public void run() { new Main().m(new Main.B2()); }
}
```

Fig. 1. Example code loading a known plugin. The Plugin does not modify `Main.main`, which ensures that the call to `main.m()` can remain inlined. If only `Plugin` is loaded, `p.run()` can also be inlined. Pre-analysing `Plugin` reveals that `a.n()` should be virtual, even if the flow analysis of the code without `Plugin` may say otherwise.

Eclipse 3.0.1. An open extensible Integrated Development Environment from `http://www.eclipse.org/`, 22858 classes from the platform and the CDT, JDT, PDE and SDK components; analysed with Sun JDK 1.4.2 for Linux, 10277 classes (using the JARs dnsns, rt, sunrsasign, jsse, jce, charsets, sunjce_provider, ldapsec and localedata).

While we have "only" two benchmarks, note that the combined size of SPECjvm98 and SPECjbb2000 is merely 11% of the size of Eclipse. Furthermore, these are the only freely available large Java systems with plugin architectures that we are aware of. Analysing benchmarks, such as the SPEC benchmarks, that do not have plugins is pointless. We are not aware of any previously published results on 0-CFA for benchmarks of this size.

We will use *app* to denote the core application together with the plugins that are available for ahead-of-time analysis. Automatically drawing a clear line between plugins and the main application is difficult considering that parts of the "core" may only be reachable from certain plugins.

Usually, flow analyses are implemented with a form of reachability built in, and more powerful powerful analyses are better at reachability. To further ensure

Jedit	Can be inlined						Cannot be inlined		Total	
	Remain valid		Can be invalidated							
			By DLCW		By DLOW not DLCW					
	app	lib	app	lib	app	lib	app	lib	app	lib
Local	682	297	0	0	0	0	20252	7808	20934	8105
CHA	682	297	69	7	18720	6178	1463	1623	20934	8105
RTA	682	297	97	51	18723	6178	1432	1579	20934	8105
TSMI	682	297	99	59	19449	7091	704	658	20934	8105
0-CFA	682	297	103	83	19592	7191	557	534	20934	8105

Eclipse	Can be inlined						Cannot be inlined		Total	
	Remain valid		Can be invalidated							
			By DLCW		By DLOW not DLCW					
	app	lib	app	lib	app	lib	app	lib	app	lib
Local	15497	472	0	0	0	0	481939	26512	497436	26984
CHA	15497	472	4105	61	366114	20796	111720	5655	497436	26984
RTA	15497	472	9024	169	366169	20797	106746	5546	497436	26984
TSMI	15497	472	11479	439	420029	23097	50431	2976	497436	26984
0-CFA	15497	472	9921	46	428944	23971	43074	2495	497436	26984

Fig. 2. Experimental results; each number is a count of virtual call sites

a fair comparison of the analyses, reachability is first done once in the same way for all analyses. Then each of the analyses is run with reachability disabled. The initial reachability analysis is based on RTA and assumes that all of *app* is live, in particular, all local plugins are treated as roots for reachability. The analysis determines the part of the library (classpath, JDK) which is live, denoted *lib*, and then we remove the remainder of the library.

The combination *app + lib* is the "closed world" that is available to the ahead-of-time compiler, in contrast to all of the code that could theoretically be dynamically loaded from the "open world". We use the abbreviations:

DLCW = Dynamic Loading from Closed World
DLOW = Dynamic Loading from Open World.

In other words, DLCW means loading a local plugin, whereas DLOW means loading a plugin from, say, the Internet.

Figure 2 shows the static number of virtual call sites that can be inlined under the respective circumstances. The numbers show that loading from the local set of plugins results in an extremely small number of possible invalidations (DLCW). The numbers also show that preanalyzing plugins is about 50% more effective for 0-CFA than for CHA: the number of additional devirtualisations is respectively 57% and 49% higher for 0-CFA after compensating for the higher number of devirtualisations of 0-CFA. When loading arbitrary code from the

open world (DLOW), the compiler has to consider almost all devirtualised call sites for invalidation. Only a tiny fraction of all virtual calls can be guaranteed to never require revirtualisation in a setting with dynamic loading—a compiler that cannot revirtualise calls can only perform a fraction of the possible inlining optimisations.

The data also shows that TSMI and 0-CFA are quite close in terms of precision, which is good news since this means it is possible to use the type-safe variant without loosing many opportunities for optimisation. As expected, using 0-CFA or TSMI instead of CHA or RTA cuts in half the number of virtual calls left in the code after optimisation. Notice that for Eclipse, in the column for call sites that can be inlined and invalidated by DLCW, 0-CFA has a *smaller* number than TSMI. This is not an anomaly; on the contrary, it shows that 0-CFA is so good that it both identifies 7357 more call sites in *app* for inlining than TSMI *and* determines that many call sites cannot be invalidated by DLCW.

The closest related work to our experiment is the extant analysis of Sreedhar, Burke, and Choi [22] which determines whether a variable can only contain objects of classes from the closed world. They did not consider the more detailed question of whether inlining can be invalidated due to DLCW or only due to DLOW. Their largest benchmark was *jess* which has 112 classes.

3 Overview of Our Framework

Our framework uses a simple construct called `dynnew` which abbreviates the Java expression `Class.forName(...).newInstance()`, that is, an operation that loads some class and immediately instantiates it. Using this construct means that we do not need to model the result of `Class.forName(...)` and deal with objects that reify classes, simplifying the operational semantics.

A New Type System. In later sections we will prove that TSMI supports type-safe method inlining for a setting with dynamic class loading. We use a new type system for the intermediate representation: each type is a pair of Java types. In this section we explain the main problem that lead us to the new type system. Our running example is an extended version of one from our paper on TSMI [11].

```
class B {
  B m() { return this; }
}

class C extends B {
  C f;
  B m() {
    return this.f;
  }
}
```

```
// code snippet 1:
B x = new C(); // x is a field
x = x.m();
x = ((B)new C()).m();

// code snippet 2:
B y;           // y is a field
if (...) { y = new C(); }
   else  { y = (B)dynnew; }
y = y.m();
```

The two code snippets contain three method calls, each to a receiver object of type `B`. CHA will for each method call determine that there are two possible target methods, namely `B.m` and `C.m`, so CHA will lead to inlining of *none* of the three call sites.

In snippet 1, which does not have dynamic loading, both of the calls have unique targets that are small code fragments, so it makes sense to inline these calls:

```
x = x.f;                          // does not type check
x = ((B)new C()).f                // does not type check
```

These two assignments do not type check because while `this` in class `C` has static type `C`, both `x` and `(B)new C()` have static type `B`. Since `B` has no `f` field, both field selections fail the type checker. As explained in our previous paper [11], we remedy this problem by changing static type information to reflect the more accurate information the flow analysis has. In particular, the flow analysis has determined that `x` and the cast expression only evaluate to objects of type `C`, and so we transform the static type information to produce the following well-typed code snippet:

```
C x = new C();
x = x.f;                          // type checks
x = ((C)new C()).f;               // type checks
```

To understand the problems introduced by dynamic class loading, let us consider code snippet 2. The method call `y.m()` has a unique target method *at least until the next dynamic class loading*. So it makes sense to inline the call, even though that decision may be invalidated later. To see how this may be achieved, the key question is:

Question: What is the flow set for `dynnew` ?

With CHA, the answer is given by the static type of `dynnew`, which is `Object`, and so the flow set is “all classes in the program”. Since `dynnew` has no impact on the execution *until the next dynamic class loading*, we could assign `dynnew` the empty flow set! We extend TSMI to dynamic loading in this way. However, this idea runs into a difficulty quickly, as we explain next.

For code snippet 2, our previous approach transforms the types in a way that preserves well-typedness:

```
C y;                          // the type of y is changed to C
if (...) { y = new C(); }
   else  { y = (C)dynnew; }  // the type cast is changed to C
y = y.m();
```

Let us now suppose that control reaches `dynnew` and that it loads and instantiates a class `D` which extends class `B` and is otherwise unrelated to class `C`. In the original code snippet 2, the cast of `dynnew` is to `B`, so it succeeds. However,

in the transformed code snippet, the cast of `dynnew` is to `C`, so it fails. Thus, if we transform the types in the style of our previous paper [11] and we do not transform the types *again* at the time of evaluating `dynnew`, we change the meaning of the program!

The source of the difficulty is that a type cast can viewed as doing double duty: it does a run-time check and it helps the type checker. Our solution is to change the cast into a form that uses a *pair* of types. In code snippet 2, we would change the cast of `dynnew` to `(B,C)dynnew`. We say that `B` is the *original* type and that `C` is the *current* type. The current type is based on the flow analysis. The original type is used to do the run-time check while the current type is used to help the type checker. In fact, we need to change the entire type system and use pairs of types everywhere, not just in casts. Note, to be sound, the current type must be a subtype of the original type.

Armed with the idea of using pairs of types, we can now state the type of `dynnew`. The original type continues to be `Object` and the current type is derived from the flow set which is the empty set. The empty set corresponds to a type which is a subtype of all other types. To reflect that, we introduce a type `Null` and give `dynnew` the type `(Object, Null)`. This has the pleasant side effect that we can remove an artificial requirement from the original formulation of TSMI, namely that all flow sets have to be nonempty.

Returning to code snippet 2, our approach will first transform the snippet into:

```
(B,C) y;                           // the type of y is changed to (B,C)
if (...) { y = new C(); }
   else  { y = (B,C)dynnew; }  // the type cast is changed to (B,C)
y = y.m();
```

Next, evaluating `dynnew` and thereby loading and instantiating a class `D` can be modeled as replacing `dynnew` with `new D()` as well as a new flow analysis of the program. The new analysis changes the current types, resulting in the following type-correct code:

```
(B,B) y;
if (...) { y = new C(); }
   else  { y = (B,D)new D(); }
y = y.m();
```

Notice that the current type of `y` was `B` initially, then the TSMI-based optimisation changed it to the more specific type `C`, and then the dynamic loading of class `D` changed the current type of `y` back to `B`.

In summary, the new ideas are:

- A type is a pair of Java types in which the second Java type is a subtype of the first Java type.
- The `Null` type is used to type `dynnew`.
- A type cast uses the first Java type in the pair.

Our main theorem is that with a type system based on those three ideas, TSMI-based devirtualisation and revirtualisation is type preserving. As our experiments in the previous section show, the new approach will lead to considerably more inlining than the previously best approach, namely CHA. Later we formalise our ideas and prove the main theorem. First we clarify how revirtualisation is done and how we formalise it, and clarify how we do our proofs.

Patch Construct. Until now we have not said much about how a virtual machine revirtualises a method invocation. The main problem with revirtualisation is that an invalidated method inlining may be in a currently executing method, requiring a nontrivial update of the program state. We focus on a technique for doing this update called *patching*, used by some virtual machines (for example [14] and ORP [5,6]). Patching is a form of in-place code modification for reverting to unoptimised code, and does not require any update of the stack or recompilation of methods. The basic idea is to compile the call `x.m()` to the following code:

```
label l1:  [Inline x.C::m()]
label l3:  ...
label l2:  x.m();     [out of line]
           jump l3;
```

(Where out of line means after the end of the function being compiled.) Then if a class is loaded that invalidates the inlining, the virtual machine writes a `jump l2;` instruction at address `l1`. There are important low-level details that we abstract (these and techniques other than patching are described in our previous paper [10]).

To formalise this idea in a small language, we need an expression of the form $\mathtt{e}_1$ $\mathtt{patchto}^\ell$ $\mathtt{e}_2$ where ℓ is a label. Additionally, program states will have a component, called the patch set, that is a set of labels of patches that have been applied. If ℓ is in this set then the above expression acts like $\mathtt{e}_2$, if not it acts like $\mathtt{e}_1$. This idea models what the assembly sequence above does.

Note that, as in previous papers, we concentrate on devirtualisation, the first step of method inlining, as the other step is straightforward. Given this focus, a general patch construct is not needed. Instead we use a construct of the form $\mathtt{e.[C::]}^\ell\mathtt{m()}$, which can be though of as $\mathtt{e.C::m()}$ $\mathtt{patchto}^\ell$ $\mathtt{e.m()}$ where `e.C::m()` invokes `C`'s implementation of `m` on `e`, and ultimately should be thought of as the code above.

The correctness of speculative inlining with patching is far less obvious than the correctness of inlining for whole programs. We use a proof framework developed in our previous paper [10]. Note that we do devirtualisation of both the initial program and of dynamically loaded classes. Furthermore, the patching operation, which is part of the optimisation, is a runtime operation. The usual formalisation methods do not suffice, and instead we formalise the optimisation as a second semantics. This semantics includes the transformation that does devirtualisation and the patching operation as part of the semantics of `dynnew`. To prove correctness of the optimisation we show that the optimising semantics

gives the same meaning to a program as a standard semantics does. To prove type preservation, we prove the optimising semantics type safe.

4 Dynamic Loading Language

This section begins the formal development of our results. It defines a simple language with dynamic class loading that is the source language for the optimisation. The language is a variant of Featherweight Java (FJ [13]), adding just one new expression form for dynamically loading a new class. Due to space limitations we omit many standard or obvious details (readers can refer to the original FJ paper or our previous dynamic loading paper). The optimised code will use a slightly different syntax (see the following section), here is the common syntax:

Expressions	$\mathtt{e} ::= \mathtt{x}^\ell \mid \mathtt{new}\ \mathtt{C}^\ell(\overline{\mathtt{e}}) \mid \mathtt{e.f}^\ell \mid \mathtt{e.m}^\ell(\overline{\mathtt{e}}) \mid (\mathtt{t})^\ell\mathtt{e} \mid \mathtt{dynnew}^\ell$
Method Declarations	$\mathtt{M} ::= \mathtt{t}^\ell\ \mathtt{m}(\overline{\mathtt{t}\ \mathtt{x}^\ell})\ \{\ \mathtt{return}\ \mathtt{e};\ \}$
Class Declarations	$\mathtt{CD} ::= \mathtt{class}\ \mathtt{C}_1\ \mathtt{extends}\ \mathtt{C}_2\ \{\ \overline{\mathtt{t}\ \mathtt{f}^\ell};\ \overline{\mathtt{M}}\ \}$

And here is the standard syntax:

Types	$\mathtt{t} ::= \mathtt{C}$
Program State	$\mathtt{P} ::= (\overline{\mathtt{CD}};\mathtt{e})$

We use standard metavariables and the bar notation from the FJ paper.

To simplify matters, we assume that field names are unique, that all $\mathtt{x}^\ell$ expressions have the same label as the binder of $\mathtt{x}$, and that all labels of `this` in a class have the same label. These restrictions mean that $lab(\mathtt{f})$ identifies a unique label for each field declared in a program, and that in the given scope $lab(\mathtt{x})$ identifies a unique label for each variable in that scope.

Some auxiliary definitions that are used in the rest of the paper appear in appendix A. The standard operation semantics is similar to FJ extended with a rule for `dynnew`:

$$\frac{\mathtt{CD} = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \cdots\ \{\ \cdots\ \}}{(\overline{\mathtt{CD}};\mathtt{X}\langle\mathtt{dynnew}^\ell\rangle) \overset{\mathtt{CD},\overline{\mathtt{e}},\ell'}{\longmapsto_{\mathtt{s}}} (\overline{\mathtt{CD}},\mathtt{CD};\mathtt{X}\langle\mathtt{new}\ \mathtt{C}^{\ell'}(\overline{\mathtt{e}})\rangle)} \quad (1)$$

Here $\mathtt{X}$ ranges over evaluation contexts. To keep the semantics deterministic, we explicitly label the reduction with a label of the form $(\mathtt{CD},\overline{\mathtt{e}},\ell)$, where $\mathtt{CD}$ is the newly loaded class, $\overline{\mathtt{e}}$ are the initialiser expressions, and ℓ is the label to use on the new object.

The typing rules are those of Featherweight Java extended with a rule for `dynnew`; they can be recovered from the more general rules in Figure 4 by ignoring the right type in the type pairs. The type system is sound as can be proven by standard techniques.

$$poly(\mathtt{P},\phi) = \{\ell \mid \mathtt{e.[C::]}^{\ell}\mathtt{m}(\overline{\mathtt{e}}) \in \mathtt{P}, \exists \mathtt{D} \in \phi(lab(\mathtt{e})) : impl(\mathtt{P},\mathtt{D},\mathtt{m}) \neq \mathtt{C::m}\}$$

$$\frac{fields(\overline{\mathtt{CD}},\mathtt{C}) = \overline{\mathtt{t}}\;\overline{\mathtt{f}};}{(\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{new}\ \mathtt{C}^{\ell_1}(\overline{\mathtt{e}}).\mathtt{f}_i^{\ell_2}\rangle) \mapsto_o (\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{e}_i\rangle)} \quad (2)$$

$$\frac{mbody(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m}) = (\overline{\mathtt{x}},\mathtt{e},\ell)}{(\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{new}\ \mathtt{C}^{\ell_1}(\overline{\mathtt{e}}).\mathtt{m}^{\ell_2}(\overline{\mathtt{d}})\rangle) \mapsto_o (\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{e}\{\mathtt{this},\overline{\mathtt{x}} := \mathtt{new}\ \mathtt{C}^{\ell_1}(\overline{\mathtt{e}}),\overline{\mathtt{d}}\}\rangle)} \quad (3)$$

$$\frac{\overline{\mathtt{CD}} \vdash \mathtt{C} <: \mathtt{D}}{(\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle(\mathtt{(D,E)})^{\ell'}\mathtt{new}\ \mathtt{C}^{\ell}(\overline{\mathtt{e}})\rangle) \mapsto_o (\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{new}\ \mathtt{C}^{\ell}(\overline{\mathtt{e}})\rangle)} \quad (4)$$

$$\frac{\begin{array}{c}\mathtt{CD} = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \cdots\ \{\ \cdots\ \} \qquad \mathtt{P} = (\overline{\mathtt{CD}},\mathtt{CD};\mathtt{S};\mathtt{X}\langle\mathtt{new}\ \mathtt{C}^{\ell}(\overline{\mathtt{e}})\rangle) \qquad \phi = fa(\mathtt{P}) \\ \overline{\mathtt{CD}'} = retype(\overline{\mathtt{CD}},\phi) \qquad \mathtt{X}' = retype(\mathtt{X},\phi) \qquad \mathtt{CD}' = [\![retype(\mathtt{CD},\phi)]\!]_{\overline{\mathtt{CD}},\mathtt{CD},\phi} \\ \overline{\mathtt{e}'} = [\![retype(\overline{\mathtt{e}},\phi)]\!]_{\overline{\mathtt{CD}},\mathtt{CD},\phi} \qquad \mathtt{S}' = \mathtt{S} \cup poly(\mathtt{P},\phi)\end{array}}{(\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{dynnew}^{\ell}\rangle) \overset{\mathtt{CD},\overline{\mathtt{e}},\ell'}{\mapsto_o} (\overline{\mathtt{CD}'},\mathtt{CD}';\mathtt{S}';\mathtt{X}'\langle\mathtt{new}\ \mathtt{C}^{\ell'}(\overline{\mathtt{e}'})\rangle)} \quad (5)$$

$$\frac{mbody\left(\overline{\mathtt{CD}},\left\{\begin{array}{ll}\mathtt{C} & \ell_2 \in \mathtt{S} \\ \mathtt{D} & \ell_2 \notin \mathtt{S}\end{array}\right\},\mathtt{m}\right) = (\overline{\mathtt{x}},\mathtt{e},\ell)}{(\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{new}\ \mathtt{C}^{\ell_1}(\overline{\mathtt{e}}).\mathtt{[D::]}^{\ell_2}\mathtt{m}(\overline{\mathtt{d}})\rangle) \mapsto_o (\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{e}\{\mathtt{this},\overline{\mathtt{x}} := \mathtt{new}\ \mathtt{C}^{\ell_1}(\overline{\mathtt{e}}),\overline{\mathtt{d}}\}\rangle)} \quad (6)$$

Fig. 3. Optimised Operational Semantics

5 Devirtualisation Optimisation

This section formalises speculative devirtualisation with patching for revirtualisation as a second semantics, called the *optimising semantics*, for the language of the previous section. The additional constructs required are described next, following by the actual transformation, and finally the semantics and the type system.

Syntax. The optimised semantics needs a patching construct and an associated patch set in the program states, and two types in each static typing annotation—the original and the current type. The modified syntax is:

Types	$\mathtt{t} ::= (\mathtt{C}_1,\mathtt{C}_2)$
Expressions	$\mathtt{e} ::= \cdots \mid \mathtt{e.[C::]}^{\ell}\mathtt{m}(\overline{\mathtt{e}})$
Program States	$\mathtt{P} ::= (\overline{\mathtt{CD}};\mathtt{S};\mathtt{e})$

Here $\mathtt{S}$, called the *patch set*, is the set of labels of the patch constructs that had to be revirtualised. A patch construct has the form $\mathtt{e.[C::]}^{\ell}\mathtt{m}(\overline{\mathtt{e}})$. If ℓ is in the patch set $\mathtt{S}$ then this expression acts like a normal virtual method invocation $\mathtt{e.m}^{\ell}(\overline{\mathtt{e}})$. Otherwise it acts like a nonvirtual method invocation—it invokes $\mathtt{C}$'s version of $\mathtt{m}$ on object $\mathtt{e}$ with arguments $\overline{\mathtt{e}}$. Types are now pairs where the left class name is the original type from the unoptimised code, and the right class name is the current type based on the current flow analysis.

Transformation. The transformation of code is based on a *flow* that assigns sets of class names, called *flow sets*, to expressions, fields, method parameters,

and method returns. The set should include all classes in the current program state that the expression might evaluate to. A flow analysis takes a program state and returns a flow for it, and it should ignore the current types. Before applying the transformation, the static type information must be transformed so that the current types reflect the flow used. The *retype* function achieves this change. Its definition is in Appendix A, as the only interesting clause is: $\mathit{retype}((\texttt{C}_1\texttt{,C}_2)^\ell, \phi) = (\texttt{C}_1\texttt{,} \sqcup \phi(\ell))$. The transformation takes an expression, method declaration, or class declaration and changes monomorphic virtual method invocations into patchable nonvirtual method invocations. It appears in Appendix A as the only interesting clause is:

$$[\![\texttt{e.m}^\ell\texttt{(}\overline{\texttt{e}}\texttt{)}]\!]_{\overline{\texttt{CD}},\phi} = [\![\texttt{e}]\!]_{\overline{\texttt{CD}},\phi}\texttt{.[C::]}^\ell\texttt{m(}[\![\overline{\texttt{e}}]\!]_{\overline{\texttt{CD}},\phi}\texttt{)} \quad \text{if } \forall \texttt{D} \in \phi(\mathit{lab}(\texttt{e})) : \mathit{impl}(\overline{\texttt{CD}}, \texttt{D}, \texttt{m}) = \texttt{C::m}$$

Optimised Semantics. The optimised semantics is parameterised by a flow analysis *fa* (that is, a function that takes an optimised-syntax program state and returns a flow for it). A standard syntax program $(\overline{\texttt{CD}}\texttt{;e})$ starts in the optimised semantics state $([\![\mathit{retype}(\overline{\texttt{CD}}, \phi)]\!]_{\overline{\texttt{CD}},\phi}\texttt{;}\emptyset\texttt{;}[\![\mathit{retype}(\texttt{e}, \phi)]\!]_{\overline{\texttt{CD}},\phi})$ where $\phi = \mathit{fa}(\overline{\texttt{CD}}\texttt{;}\emptyset\texttt{;e})$. In other words a flow analysis is performed on the initial program and used to transform it to form the initial state along with an empty patch set.

The reduction rules for the optimised semantics appear in Figure 3. The rules are similar to the standard semantics with the following modifications. The rule for cast uses the original type in the cast rather than the current type to determine if the cast should succeed. The rule for dynamic new is the most complex. It performs a flow analysis on the unoptimised new program state. Then it uses this flow analysis to retype the program state and to transform the new class declaration and initialiser expressions. Finally, it adds to the patch set the labels of patch constructs that are no longer monomorphic. The rule for the patch construct is similar to the rule for method invocation except in how it finds the method body. If the label is in the patch set, then the construct is "patched" and should act like a virtual method invocation. In this case it uses the object's class to lookup the body as in the rule for method invocation. If the label is not in the patch set, then the construct acts like a nonvirtual invocation, and uses the class in the construct, `D`, to lookup the method body.

Type System. The typing rules appear in Figure 4. The rules are fairly straightforward. They essentially are checking the original and current typing in parallel. To look up field or method types, since these are the same whether we look in the superclass or subclass, we simply use the original type. Two rules treat the current and original types differently. For dynamic new, the current is `Null` as it is always retyped before it is replaced by an actual object, but its original type must be `Object`. For the patching construct, if not currently patched then the object must be in the type `E` being dispatched to, so we require the current type to be a subtype of this.

Except for the details of subtyping, the rules are deterministic, and for a program state `P`, there is a unique `t` and derivation of $\vdash \texttt{P} \in \texttt{t}$. Therefore, given a program and an occurrence of a label in it, there is a uniquely determined type associated with that occurrence: either the type of the expression it labels, or

$$\frac{}{\overline{\texttt{CD}} \vdash \texttt{Null <: Object} \qquad \overline{\texttt{CD}} \vdash \texttt{Object <: Object}} \tag{7}$$

$$\frac{\texttt{class C extends D} \{ \cdots \} \in \overline{\texttt{CD}}}{\overline{\texttt{CD}} \vdash \texttt{Null <: C} \qquad \overline{\texttt{CD}} \vdash \texttt{C <: C} \qquad \overline{\texttt{CD}} \vdash \texttt{C <: D}} \tag{8}$$

$$\frac{\overline{\texttt{CD}} \vdash \texttt{C <: D} \qquad \overline{\texttt{CD}} \vdash \texttt{D <: E}}{\overline{\texttt{CD}} \vdash \texttt{C <: E}} \tag{9}$$

$$\frac{\overline{\texttt{CD}} \vdash \texttt{C}_2 \texttt{ <: } \texttt{C}_1}{\overline{\texttt{CD}} \vdash \texttt{(C}_1\texttt{,C}_2\texttt{)}} \tag{10}$$

$$\frac{\overline{\texttt{CD}} \vdash \texttt{C}_1 \texttt{ <: } \texttt{D}_1 \qquad \overline{\texttt{CD}} \vdash \texttt{C}_2 \texttt{ <: } \texttt{D}_2}{\overline{\texttt{CD}} \vdash \texttt{(C}_1\texttt{,C}_2\texttt{) <: (D}_1\texttt{,D}_2\texttt{)}} \tag{11}$$

$$\frac{}{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{x} \in \Gamma(\texttt{x})} \tag{12}$$

$$\frac{\mathit{fields}(\overline{\texttt{CD}}, \texttt{C}) = \overline{\texttt{t}}\ \overline{\texttt{f}}\texttt{;} \qquad \overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{t}'} \qquad \overline{\texttt{CD}} \vdash \overline{\texttt{t}'} \texttt{ <: } \overline{\texttt{t}}}{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{new C}^{\ell}\texttt{(}\overline{\texttt{e}}\texttt{)} \in \texttt{(C,C)}} \tag{13}$$

$$\frac{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{e} \in \texttt{(C,D)} \qquad \mathit{fields}(\overline{\texttt{CD}}, \texttt{C}) = \overline{\texttt{t}}\ \overline{\texttt{f}}\texttt{;}}{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{e.f}_i^{\ell} \in \texttt{t}_i} \tag{14}$$

$$\frac{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{e} \in \texttt{(C,D)} \qquad \mathit{mtype}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = \overline{\texttt{t}} \to \texttt{t} \qquad \overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{t}'} \qquad \overline{\texttt{CD}} \vdash \overline{\texttt{t}'} \texttt{ <: } \overline{\texttt{t}}}{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{e.m}^{\ell}\texttt{(}\overline{\texttt{e}}\texttt{)} \in \texttt{t}} \tag{15}$$

$$\frac{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{e} \in \texttt{t}' \qquad \overline{\texttt{CD}} \vdash \texttt{t}}{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{(t)}^{\ell}\texttt{e} \in \texttt{t}} \tag{16}$$

$$\frac{}{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{dynnew}^{\ell} \in \texttt{(Object,Null)}} \tag{17}$$

$$\frac{\begin{array}{c}\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{e} \in \texttt{(C,D)} \\ \mathit{mtype}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = \overline{\texttt{t}} \to \texttt{t} \\ \overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{t}'} \\ \overline{\texttt{CD}} \vdash \overline{\texttt{t}'} \texttt{ <: } \overline{\texttt{t}} \\ \mathit{mtype}(\overline{\texttt{CD}}, \texttt{E}, \texttt{m}) \text{ is defined} \\ \ell \notin \texttt{S} \Rightarrow \overline{\texttt{CD}} \vdash \texttt{D <: E}\end{array}}{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{e.[E::]}^{\ell}\texttt{m(}\overline{\texttt{e}}\texttt{)} \in \texttt{t}} \tag{18}$$

$$\frac{\begin{array}{c}\overline{\texttt{CD}} \vdash \texttt{t} \qquad \overline{\texttt{CD}} \vdash \overline{\texttt{t}} \\ \overline{\texttt{CD}}; \texttt{S}; \texttt{this} : \texttt{(C,C)}, \overline{\texttt{x}} : \overline{\texttt{t}} \vdash \texttt{e} \in \texttt{t}' \qquad \overline{\texttt{CD}} \vdash \texttt{t}' \texttt{ <: t} \\ \mathit{can\text{-}declare}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}, \overline{\texttt{t}} \to \texttt{t})\end{array}}{\overline{\texttt{CD}}; \texttt{S} \vdash \texttt{t}^{\ell}\ \texttt{m(}\overline{\texttt{t}}\ \overline{\texttt{x}^{\ell}}\texttt{)} \{ \texttt{return e;} \} \text{ in } \texttt{C}} \tag{19}$$

$$\frac{\overline{\texttt{CD}} \vdash \overline{\texttt{t}} \qquad \overline{\texttt{CD}}; \texttt{S} \vdash \overline{\texttt{M}} \text{ in } \texttt{C}}{\overline{\texttt{CD}}; \texttt{S} \vdash \texttt{class C extends D} \{ \overline{\texttt{t}}\ \overline{\texttt{f}^{\ell}}\texttt{;}\ \overline{\texttt{M}} \}} \tag{20}$$

$$\frac{\overline{\texttt{CD}}; \texttt{S} \vdash \overline{\texttt{CD}} \qquad \overline{\texttt{CD}}; \texttt{S}; \cdot \vdash \texttt{e} \in \texttt{t}}{\vdash \texttt{(}\overline{\texttt{CD}}\texttt{;S;e)} \in \texttt{t}} \tag{21}$$

Fig. 4. Typing Rules for the Optimised Syntax

the field, return, or parameter type that it labels. A flow ϕ for a program is *type respecting* if and only if for each label ℓ in the program, each class `C` in $\phi(\ell)$, and each original type `D` associated with ℓ, `C` is a subtype of `D`.

6 Correctness

In this section we prove the optimisation correct, that is, that it preserves typability and operational semantics. The optimisation is correct, however, only for certain flow analyses—the ones that respect the typing rules and approximate the operational semantics. A flow ϕ for a program $\mathtt{P}$ is *acceptable* exactly when it satisfies the conditions in Figure 5. A flow analysis *fa* is correct if *fa*($\mathtt{P}$) is an acceptable and type-respecting flow for $\mathtt{P}$ whenever $\vdash \mathtt{P} \in \mathtt{t}$ for some $\mathtt{t}$. We prove the optimisation correct when it is based on a correct flow analysis.

Typability Preservation. Since the optimisation is stated as a second semantics for the language, typability preservation means that a well-typed standard syntax program does not get stuck in the optimised semantics. However, it is not enough that the original program type checks, all dynamically loaded classes must type check as well. We say that $(\mathtt{CD}, \overline{\mathtt{e}}, \ell)$ type checks with respect to program $(\overline{\mathtt{CD}};\mathtt{S};\mathtt{e})$ exactly when $\overline{\mathtt{CD}}, \mathtt{CD}; \mathtt{S} \vdash \mathtt{CD}$ and $\overline{\mathtt{CD}}, \mathtt{CD}; \mathtt{S}; \cdot \vdash \overline{\mathtt{e}} \in \overline{\mathtt{t}}$ where $\mathtt{CD} = \mathtt{class\ C\ extends}\ \cdots\ \{\ \cdots\ \}$ and *fields*$(\overline{\mathtt{CD}}, \mathtt{CD}, \mathtt{C}) = \overline{\mathtt{t}}\ \overline{\mathtt{f}}\mathtt{;}$. We say that a reduction sequence type checks exactly when the initial program state type checks and all the labels in the reduction sequence type check with respect to the program state that precedes them.

Theorem 1 (Typability Preservation). *If* $\mathtt{P}$ *is a well-typed standard-syntax program, then any well-typed reduction sequence in the optimised semantics, which starts from a state corresponding to* $\mathtt{P}$ *and is based on a correct flow analysis, does not end in a stuck state.*

The proof is given in the full version of the paper, which is available from the webpage `http://www.cs.ucla.edu/~palsberg/publications.html`. The key to proving the theorem is proving that at each point in the reduction sequence the program state type checks and there is an acceptable and type-respecting flow for the program state. Formally, we define $\vdash (\mathtt{P}, \phi)\ \mathsf{good}$ to mean $\vdash \mathtt{P} \in \mathtt{t}$ for some $\mathtt{t}$, ϕ is an acceptable and type-respecting flow for $\mathtt{P}$, and the current type of every static typing annotation in $\mathtt{P}$ is $\sqcup\phi(\ell)$ where ℓ is the label associated with the annotation. As with standard type soundness arguments, we show that reduction preserves goodness (rather than typability), and that typable (a subset of good) states are not stuck.

Operational Correctness. We prove that the optimisation preserves semantics, specifically that the optimised semantics simulates the standard semantics and vice versa. To state the result we need a *correspondence* relation *corresponds*$_\phi$ $(\mathtt{P}, \mathtt{P}')$. This relation generalises the transformation slightly to reflect the fact that the transformation is applied at consecutive loading points rather than all at once. Its definition appears in the full version of the paper. Essentially, where the left program has a virtual dispatch the right program may have one of two expressions. It can have a corresponding virtual dispatch. It can also have an equivalent patch construct if the virtual dispatch is monomorphic in the current

– For each $\texttt{new C}^\ell\texttt{(}\overline{\texttt{e}}\texttt{)}$ in $\texttt{P}$ where $\mathit{fields}(\overline{\texttt{CD}}, \texttt{C}) = \overline{\texttt{t}}\ \overline{\texttt{f}}\texttt{;}$:

$$\phi(\mathit{lab}(\overline{\texttt{e}})) \subseteq \phi(\mathit{lab}(\overline{\texttt{f}})) \tag{22}$$
$$\texttt{C} \in \phi(\ell) \tag{23}$$

– For each $\texttt{e.f}^\ell$ in $\texttt{P}$:

$$\phi(\mathit{lab}(\texttt{f})) = \phi(\ell) \tag{24}$$

– For each $\texttt{e.m}^\ell\texttt{(}\overline{\texttt{e}}\texttt{)}$ in $\texttt{P}$ where $\texttt{e}$ has type $\texttt{(C}_1\texttt{,C}_2\texttt{)}$ and $\mathit{mbody}(\texttt{P}, \texttt{C}_1, \texttt{m}) = (\overline{\texttt{x}}, \texttt{e}', \ell')$:

$$\phi(\mathit{lab}(\overline{\texttt{e}})) \subseteq \phi(\mathit{lab}(\overline{\texttt{x}})) \tag{25}$$
$$\phi(\ell') = \phi(\ell) \tag{26}$$

And for each $\texttt{D} \in \phi(\mathit{lab}(\texttt{e}))$, $\mathit{impl}(\texttt{P}, \texttt{D}, \texttt{m}) = \texttt{E::m}$, and ℓ' labels $\texttt{this}$ in $\texttt{E}$:

$$\phi(\mathit{lab}(\texttt{e})) \subseteq \phi(\ell') \tag{27}$$

– For each $\texttt{((C,D))}^\ell\texttt{e}$ in $\texttt{P}$:

$$\phi(\mathit{lab}(\texttt{e})) \cap \mathit{subclasses}(\texttt{P}, \texttt{C}) \subseteq \phi(\ell) \tag{28}$$

– For each $\texttt{dynnew}^\ell$ in $\texttt{P}$:

$$\phi(\ell) = \emptyset \tag{29}$$

– For each $\texttt{e.[C::]}^\ell\texttt{m(}\overline{\texttt{e}}\texttt{)}$ in $\texttt{P}$ where $\texttt{e}$ has type $\texttt{(C}_1\texttt{,C}_2\texttt{)}$ and $\mathit{mbody}(\texttt{P}, \texttt{C}_1, \texttt{m}) = (\overline{\texttt{x}}, \texttt{e}', \ell')$:

$$\phi(\mathit{lab}(\overline{\texttt{e}})) \subseteq \phi(\mathit{lab}(\overline{\texttt{x}})) \tag{30}$$
$$\phi(\ell') = \phi(\ell) \tag{31}$$

And if $\ell \in \texttt{S}$ where $\texttt{P} = (\cdots\texttt{;S;}\cdots)$ then for each $\texttt{D} \in \phi(\mathit{lab}(\texttt{e}))$, $\mathit{impl}(\texttt{P}, \texttt{D}, \texttt{m}) = \texttt{E::m}$, and ℓ' labels $\texttt{this}$ in $\texttt{E}$:

$$\phi(\mathit{lab}(\texttt{e})) \subseteq \phi(\ell') \tag{32}$$

And if $\ell \notin \texttt{S}$ then the following where $\mathit{impl}(\texttt{P}, \texttt{C}, \texttt{m}) = \texttt{E::m}$ and ℓ' labels $\texttt{this}$ in $\texttt{E}$:

$$\phi(\mathit{lab}(\texttt{e})) \subseteq \phi(\ell') \tag{33}$$

– For each class $\texttt{C}$ in $\texttt{P}$ with label ℓ for $\texttt{C}$'s $\texttt{this}$ occurrences:

$$\texttt{C} \in \phi(\ell) \tag{34}$$

– For each method $\texttt{t}^\ell\ \texttt{m(}\overline{\texttt{t}}\ \overline{\texttt{x}^\ell}\texttt{)}\ \{\ \texttt{return e;}\ \}$ in $\texttt{P}$:

$$\phi(\mathit{lab}(\texttt{e})) \subseteq \phi(\ell) \tag{35}$$

– If $\texttt{t}^{\ell_1}\ \texttt{m(}\bar{t}\ \overline{x_1^{\ell_1}}\texttt{)}\ \{\ \texttt{return e}_1\texttt{;}\ \}$ overrides $\texttt{t}^{\ell_2}\ \texttt{m(}\bar{t}\ \overline{x_2^{\ell_2}}\texttt{)}\ \{\ \texttt{return e}_2\texttt{;}\ \}$ in $\texttt{P}$ then:

$$\phi(\ell_1) = \phi(\ell_2) \tag{36}$$
$$\phi(\overline{\ell_1}) = \phi(\overline{\ell_2}) \tag{37}$$

Fig. 5. The Conditions for an Acceptable Flow Analysis

program (the subscripts $\overline{\texttt{CD}}$ and ϕ on the relation) or if the patch label is in the current patch set (the subscript $\texttt{S}$ on the relation).

Given the correspondence relation, two facts are true. First, if $\texttt{P}'$ is the initial state in the optimised semantics for program $\texttt{P}$ then $corresponds_{\phi}(\texttt{P}, \texttt{P}')$ where ϕ is the flow analysis used to compute the initial state. Second, the optimised semantics simulates the standard semantics and vice versa, as stated in the following theorem.

Theorem 2 (Operational Correctness).
If $corresponds_{\phi_1}(\texttt{P}_1, \texttt{P}'_1)$ *and the flow-analysis is correct then:*

- *If* $\texttt{P}_1 \overset{L}{\mapsto}_{\texttt{s}} \texttt{P}_2$ *then* $\texttt{P}'_1 \overset{L}{\mapsto}_{\texttt{o}} \texttt{P}'_2$ *and* $corresponds_{\phi_2}(\texttt{P}_2, \texttt{P}'_2)$ *for some* $\texttt{P}'_2$ *and* ϕ_2*.*
- *If* $\texttt{P}'_1 \overset{L}{\mapsto}_{\texttt{o}} \texttt{P}'_2$ *then* $\texttt{P}_1 \overset{L}{\mapsto}_{\texttt{s}} \texttt{P}_2$ *and* $corresponds_{\phi_2}(\texttt{P}_2, \texttt{P}'_2)$ *for some* $\texttt{P}_2$ *and* ϕ_2*.*

The proof of both these facts is very similar to the proof in our previous paper [10].

7 Conclusion

We presented a new type system and theorem that shows that TSMI is type preserving in the presence of dynamic class loading. Our experimental results show that TSMI leads to considerably more inlining than the current best approach, namely CHA. Our experimental results also show the value of analyzing all local plugins at start-up time: only few inlinings will be invalidated when loading a local plugin. The flow analysis has to be recomputed only when a plugin is loaded from non-local sources. Since such remote operations involve considerable delay anyway, the extra delay from redoing the flow analysis is unlikely to be noticable.

Researchers have recently developed many new ideas for efficiently doing flow analysis, virtualisation, and devirtualisation in JIT compilers [4,12,19,20]. Our results can form the basis of a new generation of typed intermediate representations used by powerful, type-preserving JIT compilers.

In future work we would like to go beyond the static counts of virtual call sites. We would like to count how many times each call site is executed, and count how many call sites turn out to be monomorphic at run time. Researchers might also explore how our results fit with recent work on dynamic code updates [23].

References

1. B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno virtual machine. *IBM System Journal*, 39(1), February 2000.

2. David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of OOPSLA'96, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, 1996.
3. David Francis Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, Computer Science Division, University of California, Berkeley, December 1997. Report No. UCB/CSD-98-1017.
4. Jeff Bogda and Ambuj K. Singh. Can a shape analysis work at run-time? In *Java Virtual Machine Research and Technology Symposium*, 2001.
5. Michal Cierniak, Marsha Eng, Neal Glew, Brian Lewis, and James Stichnoth. The open runtime platform: A flexible high-performance managed runtime environment. *Intel Technical Journal*, 7(1), February 2003.
6. Michal Cierniak, Guei-Yuan Lueh, and James Stichnoth. Practicing judo: Java under dynamic optimizations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–26, June 2000.
7. J. Dean and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. Technical Report 94-12-01, Department of Computer Science, University of Washington at Seattle, December 1994.
8. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP'95)*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.
9. Neal Glew. An efficient class and object encoding. In *Proceedings of OOPSLA'00, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 311–324, 2000.
10. Neal Glew and Jens Palsberg. Method inlining, dynamic class loading, and type soundness. *Journal of Object Technology.* Preliminary version in Sixth Workshop on Formal Techniques for Java-like Programs, Oslo, Norway, June 2004.
11. Neal Glew and Jens Palsberg. Type-safe method inlining. *Science of Computer Programming*, 52:281–306, 2004. Preliminary version in Proceedings of ECOOP'02, European Conference on Object-Oriented Programming, pages 525–544, Springer-Verlag (*LNCS* 2374), Malaga, Spain, June 2002.
12. Martin Hirzel, Amer Diwan, and Michael Hind. Pointer analysis in the presence of dynamic class loading. In *Proceedings of ECOOP'04, 16th European Conference on Object-Oriented Programming*, pages 96–122, 2004.
13. Atsushi Igarashi, Benjamion Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 132–146, Denver, CO, USA, October 1999.
14. Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a Java just-in-time compiler. In *Proceedings of OOPSLA'00, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 294-310, 2000.
15. Christopher League, Zhong Shao, and Valery Trifonov. Representing Java classes in a typed intermediate language. In *Proceedings of ICFP '99, ACM SIGPLAN International Conference on Functional Programming*, pages 183–196, 1999.
16. Greg Morrisett, David Tarditi, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. The TIL/ML compiler: Performance and safety through types. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, 1996.

17. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.
18. Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proceedings of OOPSLA'91, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, 1991.
19. Igor Pechtchanski and Vivek Sarkar. Dynamic optimistic interprocedural analysis: A framework and an application. In *Proceedings of OOPSLA'01, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 195–210, 2001.
20. Feng Qian and Laurie J. Hendren. Towards dynamic interprocedural analysis in JVMs. In *Virtual Machine Research and Technology Symposium*, pages 139–150, 2004.
21. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, CMU, May 1991. CMU–CS–91–145.
22. Vugranam Sreedhar, Michael Burke, and Jong-Deok Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Proceedings of PLDI'00, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–207, 2000.
23. Gareth Stoyle, Michael W. Hicks, Gavin M. Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. In *Proceedings of POPL'05, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 183–194, 2005.
24. David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, USA, May 1996. ACM Press.
25. P.Y. Wei. A brief overview of the VIOLA engine, and its applications. `http://www.xcf.berkeley.edu/~wei/viola/violaIntro.html`, 1994.
26. Andrew Wright, Suresh Jagannathan, Cristian Ungureanu, and Aaron Hertzmann. Compiling Java to a typed lambda-calculus: A preliminary report. In *ACM Workshop on Types in Compilation*, Kyoto, Japan, March 1998.

Appendix A: Details of the Formalisation

The function $\mathit{fields}(\overline{\mathtt{CD}}, \mathtt{C})$ returns $\mathtt{C}$'s fields (declared and inherited) and their types; $\mathit{mtype}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m})$ returns the signature of $\mathtt{m}$ in $\mathtt{C}$, it has the form $\overline{\mathtt{t}} \to \mathtt{t}$ where $\overline{\mathtt{t}}$ are the argument types and $\mathtt{t}$ is the return type; $\mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m})$ returns the implementation of $\mathtt{m}$ in $\mathtt{C}$, it has the form $(\overline{\mathtt{x}}, \mathtt{e}, \ell)$ where $\mathtt{e}$ is the expression to evaluate, $\overline{\mathtt{x}}$ are the parameters, and ℓ is the label of the method return; $\mathit{impl}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m})$ returns the class from which $\mathtt{C}$ inherits $\mathtt{m}$ (this could be $\mathtt{C}$ itself), it has the form $\mathtt{D::m}$ where $\mathtt{D}$ is the class; $\mathit{can\text{-}declare}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}, \overline{\mathtt{t}} \to \mathtt{t})$ checks that $\mathtt{C}$ is allowed to declare $\mathtt{m}$ with signature $\overline{\mathtt{t}} \to \mathtt{t}$—this would not be the case if one of $\mathtt{C}$'s ancestors in the class hierarchy also declared $\mathtt{m}$ with a different signature.

Field Lookup, Method Information and Inheritance Checking

$$\frac{}{\mathit{fields}(\overline{\mathtt{CD}}, \mathtt{Object}) = \cdot} \qquad \frac{\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ \overline{\mathtt{t}\ \mathtt{f}^{\ell}};\ \overline{\mathtt{M}}\ \} \quad \mathit{fields}(\overline{\mathtt{CD}}, \mathtt{D}) = \overline{\mathtt{t}'\ \mathtt{f}'};}{\mathit{fields}(\overline{\mathtt{CD}}, \mathtt{C}) = \overline{\mathtt{t}'\ \mathtt{f}'}; \overline{\mathtt{t}\ \mathtt{f}};}$$

$$\frac{\overline{\texttt{CD}}(\texttt{C}) = \texttt{class C extends D} \; \{\; \overline{\texttt{t}}\; \overline{\texttt{f}}^{\overline{\ell}};\; \overline{\texttt{M}}\; \} \qquad \texttt{t}^{\ell}\; \texttt{m}(\overline{\texttt{t}}\; \overline{\texttt{x}}^{\overline{\ell}})\; \{\; \texttt{return e;}\; \} \in \overline{\texttt{M}}}{\begin{array}{c} \mathit{mtype}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = \overline{\texttt{T}} \to \texttt{t} \\ \mathit{mbody}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = (\overline{\texttt{x}}, \texttt{e}, \ell) \\ \mathit{impl}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = \texttt{C::m} \end{array}}$$

$$\frac{\overline{\texttt{CD}}(\texttt{C}) = \texttt{class C extends D} \; \{\; \overline{\texttt{t}}\; \overline{\texttt{f}}^{\overline{\ell}};\; \overline{\texttt{M}}\; \} \qquad \texttt{m} \text{ not defined in } \overline{\texttt{M}}}{\begin{array}{c} \mathit{mtype}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = \mathit{mtype}(\overline{\texttt{CD}}, \texttt{D}, \texttt{m}) \\ \mathit{mbody}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = \mathit{mbody}(\overline{\texttt{CD}}, \texttt{D}, \texttt{m}) \\ \mathit{impl}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = \mathit{impl}(\overline{\texttt{CD}}, \texttt{D}, \texttt{m}) \end{array}}$$

$$\frac{\overline{\texttt{CD}}(\texttt{C}) = \texttt{class C extends D} \; \{\; \cdots \;\} \quad \mathit{mtype}(\overline{\texttt{CD}}, \texttt{D}, \texttt{m}) = \overline{\texttt{t}'} \to \texttt{t}' \text{ implies } \overline{\texttt{t}} = \overline{\texttt{t}'} \wedge \texttt{t} = \texttt{t}'}{\mathit{can\text{-}declare}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}, \overline{\texttt{t}} \to \texttt{t})}$$

The Retyping Function and the Transformation

$$\begin{array}{ll}
\mathit{retype}((\texttt{C}_1,\texttt{C}_2)^{\ell}, \phi) & = (\texttt{C}_1,\sqcup\phi(\ell)) \\[1em]
\mathit{retype}(\texttt{x}^{\ell}, \phi) & = \texttt{x}^{\ell} \\
\mathit{retype}(\texttt{new C}^{\ell}(\overline{\texttt{e}}), \phi) & = \texttt{new C}^{\ell}(\mathit{retype}(\overline{\texttt{e}}, \phi)) \\
\mathit{retype}(\texttt{e.f}^{\ell}, \phi) & = \mathit{retype}(\texttt{e}, \phi)\texttt{.f}^{\ell} \\
\mathit{retype}(\texttt{e.m}^{\ell}(\overline{\texttt{e}}), \phi) & = \mathit{retype}(\texttt{e}, \phi)\texttt{.m}^{\ell}(\mathit{retype}(\overline{\texttt{e}}, \phi)) \\
\mathit{retype}((\texttt{t})^{\ell}\texttt{e}, \phi) & = (\mathit{retype}(\texttt{t}^{\ell}, \phi))^{\ell}\mathit{retype}(\texttt{e}, \phi) \\
\mathit{retype}(\texttt{dynnew}^{\ell}, \phi) & = \texttt{dynnew}^{\ell} \\
\mathit{retype}(\texttt{e.[C::]}^{\ell}\texttt{m}(\overline{\texttt{e}}), \phi) & = \mathit{retype}(\texttt{e}, \phi)\texttt{.[C::]}^{\ell}\texttt{m}(\mathit{retype}(\overline{\texttt{e}}, \phi)) \\[1em]
\mathit{retype}(\texttt{t}^{\ell}\; \texttt{m}(\overline{\texttt{t}}\; \overline{\texttt{x}}^{\overline{\ell}})\; \{\; \texttt{return e;}\; \}, \phi) & = \mathit{retype}(\texttt{t}^{\ell}, \phi)^{\ell}\; \texttt{m}(\mathit{retype}(\overline{\texttt{t}}^{\overline{\ell}}, \phi)\; \overline{\texttt{x}}^{\overline{\ell}}) \\
 & \quad \{\; \texttt{return } \mathit{retype}(\texttt{e}, \phi);\; \} \\
\mathit{retype}(\texttt{class C}_1 \texttt{ extends C}_2 \; \{\; \overline{\texttt{t}}\; \overline{\texttt{f}}^{\overline{\ell}};\; \overline{\texttt{M}}\; \}, \phi) & = \texttt{class C}_1 \texttt{ extends C}_2 \\
 & \quad \{\; \mathit{retype}(\overline{\texttt{t}}^{\overline{\ell}}, \phi)\; \overline{\texttt{f}}^{\overline{\ell}};\; \mathit{retype}(\overline{\texttt{M}}, \phi)\; \} \\
[\![\texttt{x}^{\ell}]\!]_{\overline{\texttt{CD}},\phi} & = \texttt{x}^{\ell} \\
[\![\texttt{new C}^{\ell}(\overline{\texttt{e}})]\!]_{\overline{\texttt{CD}},\phi} & = \texttt{new C}^{\ell}([\![\overline{\texttt{e}}]\!]_{\overline{\texttt{CD}},\phi}) \\
[\![\texttt{e.f}^{\ell}]\!]_{\overline{\texttt{CD}},\phi} & = [\![\texttt{e}]\!]_{\overline{\texttt{CD}},\phi}\texttt{.f}^{\ell} \\
[\![\texttt{e.m}^{\ell}(\overline{\texttt{e}})]\!]_{\overline{\texttt{CD}},\phi} & = [\![\texttt{e}]\!]_{\overline{\texttt{CD}},\phi}\texttt{.[C::]}^{\ell}\texttt{m}([\![\overline{\texttt{e}}]\!]_{\overline{\texttt{CD}},\phi}) \\
 & \quad \text{if } \forall \texttt{D} \in \phi(\mathit{lab}(\texttt{e})) : \mathit{impl}(\overline{\texttt{CD}}, \texttt{D}, \texttt{m}) = \texttt{C::m} \\
[\![\texttt{e.m}^{\ell}(\overline{\texttt{e}})]\!]_{\overline{\texttt{CD}},\phi} & = [\![\texttt{e}]\!]_{\overline{\texttt{CD}},\phi}\texttt{.m}^{\ell}([\![\overline{\texttt{e}}]\!]_{\overline{\texttt{CD}},\phi}) \\
 & \quad \text{otherwise} \\
[\![(\texttt{t})^{\ell}\texttt{e}]\!]_{\overline{\texttt{CD}},\phi} & = (\texttt{t})^{\ell}[\![\texttt{e}]\!]_{\overline{\texttt{CD}},\phi} \\
[\![\texttt{dynnew}^{\ell}]\!]_{\overline{\texttt{CD}},\phi} & = \texttt{dynnew}^{\ell} \\
[\![\texttt{e.[C::]}^{\ell}\texttt{m}(\overline{\texttt{e}})]\!]_{\overline{\texttt{CD}},\phi} & = [\![\texttt{e}]\!]_{\overline{\texttt{CD}},\phi}\texttt{.[C::]}^{\ell}\texttt{m}([\![\overline{\texttt{e}}]\!]_{\overline{\texttt{CD}},\phi}) \\[1em]
[\![\texttt{t}^{\ell}\; \texttt{m}(\overline{\texttt{t}}\; \overline{\texttt{x}}^{\overline{\ell}})\; \{\; \texttt{return e;}\; \}]\!]_{\overline{\texttt{CD}},\phi} & = \texttt{t}^{\ell}\; \texttt{m}(\overline{\texttt{t}}\; \overline{\texttt{x}}^{\overline{\ell}})\; \{\; \texttt{return } [\![\texttt{e}]\!]_{\overline{\texttt{CD}},\phi};\; \} \\
[\![\texttt{class C}_1 \texttt{ extends C}_2 \; \{\; \overline{\texttt{t}}\; \overline{\texttt{f}}^{\overline{\ell}};\; \overline{\texttt{M}}\; \}]\!]_{\overline{\texttt{CD}},\phi} & = \texttt{class C}_1 \texttt{ extends C}_2 \; \{\; \overline{\texttt{t}}\; \overline{\texttt{f}}^{\overline{\ell}};\; [\![\overline{\texttt{M}}]\!]_{\overline{\texttt{CD}},\phi}\; \}
\end{array}$$

Using Dependent Types to Certify the Safety of Assembly Code*

Matthew Harren and George C. Necula

Computer Science Division, University of California
Berkeley, CA, USA 94720-1776
{matth, necula}@cs.berkeley.edu

Abstract. There are many source-level analyses or instrumentation tools that enforce various safety properties. In this paper we present an infrastructure that can be used to check independently that the assembly output of such tools has the desired safety properties. By working at assembly level we avoid the complications with unavailability of source code, with source-level parsing, and we certify the code that is actually deployed.

The novel feature of the framework is an extensible dependently-typed framework that supports type inference and mutation of dependent values in memory. The type system can be extended with new types as needed for the source-level tool that is certified. Using these dependent types, we are able to express the invariants enforced by CCured, a source-level instrumentation tool that guarantees type safety in legacy C programs. We can therefore check that the x86 assembly code resulting from compilation with CCured is in fact type-safe.

1 Introduction

There are numerous ongoing efforts to design static analyses or instrumentation tools to ensure various safety and security properties of programs. In most cases, there is no independent way to ensure that the analysis or instrumentation tool was actually run on a given program. Since most of today's software security tools operate only on source code, a concerned user must obtain the source for the program in question, must run the tool himself, and is forced to trust that the tool and the compiler are working as advertised. In this paper, we describe our efforts to develop an independent verification strategy for static analyses and instrumentation tools.

A well-known example of the strategy that we advocate is the verification of type safety in Java and .NET bytecode. A compiler verifies that the original source code is type-safe, and uses this typing information to generate typed

* This research was supported in part by the National Science Foundation under grant number CCR-00225610. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 155–170, 2005.
© Springer-Verlag Berlin Heidelberg 2005

bytecode. The bytecode can then be checked for safety independently from the source code. We want to push this strategy to lower-level languages, such as assembly, and to allow more language-based enforcement tools to make use of it. Working at the assembly-language level makes our technique fit well in the current standard object-code distribution process. Furthermore, it does not require the program source code, is applicable to more source languages, and eliminates the compiler from the trusted computing base.

An additional goal of our work is to make it relatively easy for tool writers to customize a generic certification infrastructure with the rules and invariants that should hold in the processed code. To this end, the certification infrastructure performs many operations that are likely to be needed across a variety of enforcement tools.

1.1 Motivation

This work was initially motivated by requests from CCured users to have independent verification that libraries or object files have been processed by CCured. CCured [1] is a source-to-source translator that guarantees type safety for legacy C code by inserting run-time checks before potentially unsafe operations. Where necessary, it modifies data structures to accommodate metadata such as array-bound information. CCured performs extensive static analysis to minimize the changes to data structures and the number of run-time checks necessary. CCured also has many different kinds of run-time checks, for arrays, pointers on the stack, or type hierarchies. A framework that can keep up with CCured's analysis and run-time checks would be suitable for certifying the result of simpler tools such as Cqual [2] and Stackguard [3]. We believe our framework is general enough to be used with languages other than C and safety policies other than type safety.

We cannot use standard Typed Assembly Languages [4] to encode the output of CCured for two main reasons. First, the instrumentation scheme used by CCured requires dependent types to encode, for example, that a field in a structure is a pointer to a memory area whose length is stored in another field. The DTAL [5] language is dependently-typed and is at the assembly-language level, but does not allow mutation of dependently-typed records. The ability to overwrite dependent memory locations is crucial for CCured, because most C programs store pointer values in memory. We propose in this paper a new dependently-typed language that allows mutable records, by allowing the dependent-type invariants to be temporarily broken inside a basic block.

The other major obstacle in using one of the existing typed assembly languages is that it would require a special compiler that produces the desired language. Instead, we want to apply this strategy even to source-to-source transformations, in which the output of the tool is compiled using an off-the-shelf compiler. The challenge posed by an external compiler is that register allocation and other optimizations will cause us to lose the correspondence between local variables in our source code and registers in the compiled code.

Our framework relies on (untrusted) annotations for function signatures and types of global variables. These annotations are generated by the source-level

tool whose policy we enforce. We decided against using such annotations for individual program points inside of a function's body, in order to reduce sensitivity to optimizations or compilation details. Instead, we use type inference to rediscover the types of the registers and stack slots in assembly code. Our use of abstract interpretation for type inference is similar to that used in bytecode verifiers, or to that described described by Chang et al. for compiler debugging [6]. For space reasons, we do not discuss type inference in this paper.

The contributions of this paper include:

- An expressive yet practical dependent type system for low-level code that supports mutable records. We describe in Section 2 the mechanism used for customizing the type system to new policies, and present the type system itself in Section 3.
- A description of the typechecking algorithm for this type system.
- An encoding of the safety constraints of CCured in this type system, with support for arrays, dynamic typing, and stack-allocated variables whose address is taken (Section 4). We describe in Section 5 our experience using a prototype verifier that can check the CCured output for type safety.

2 Type Policies

Our type system is parameterized by a *type policy* that describes the invariants enforced by the safety tool you wish to use (CCured, for example). Factoring our type system in this way provides modularity and allows us to support extension to different safety tools. Furthermore, it lets us focus this paper on the specific contributions of our framework, such as mutable dependent types and the infrastructure for type checking.

A type policy consists of the following:

- A finite set $\mathbb{T}$ of type constructors C. These constructors are used to build policy-specific types for word-sized values, as described below.
- A subtyping relation `IsSubtype` : $\tau \rightarrow \tau \rightarrow Bool$ for the types generated by these constructors, and the associated upper bound function `TJoin` : $\tau \rightarrow \tau \rightarrow \tau$ that returns a supertype of its arguments.
- An operation `ArithType` : $\tau \rightarrow op \rightarrow \tau \rightarrow \tau$ that assigns a type to the result of binary operators given the type of the operands, and an operation `ConstType` : $const \rightarrow \tau$ that gives a type to each constant.
- A `Constrain` operation that refines a typing context after a certain boolean expression has been tested to be true.

For example, a type policy could define a type constructor "Int" for integers that will fit in a machine word, and a constructor "MaybeNullPtr σ" for possibly-NULL pointers to records with type σ. We'll see below that the framework defines the "Ptr σ" type to describe pointers to σ. Then the policy will likely define both `IsSubtype`(Ptr σ, MaybeNullPtr σ) and `IsSubtype`(MaybeNullPtr σ, Int) to be `true`. Additionally, the policy might define `ArithType`(Ptr σ, "-", Ptr σ) to be

Int. Finally, the definition of `Constrain` for this policy may promote one or more values of type MaybeNullPtr σ to Ptr σ following an appropriate NULL-check.

We defer the more detailed discussion of the `IsSubtype`, `TJoin`, `ArithType`, and `Constrain` operators until the presentation of our typechecking algorithm in Section 3.1.

Although we currently trust the soundness of the type policy, our implementation is designed to facilitate formal proofs of the soundness of verification. Such a proof would rely on lemmas that the operators of the type policy are sound with respect to the definition of the type constructors.

3 Our Type System

We describe in this section our framework for dependent types, and show how a program can be typechecked with respect to a given type policy.

Figure 1 shows the language of memory types in our framework. Field types t describe the contents of a word in memory or in a register whereas σ types describe a mutable record consisting of a sequence of related fields.

field types	$t ::= C(d_1, \dots d_n) \mid \text{Ptr } \sigma$
dependencies	$d ::= c \mid s.i \mid s$
record types	$\sigma ::= \texttt{Rec}_s.\langle 0 : t_0;\ \dots\ ;\ n-1 : t_{n-1}\rangle$
constants	c
type constructors	$C \ \in\ \mathbb{T}$

Fig. 1. The types that are assigned to registers and memory locations

The type of a word-sized location is either the instantiation of a type constructor C (given by the type policy) or a pointer to a mutable record. We saw above a few examples of nullary constructors for non-dependent types; constructors for dependent types are parameterized on one or more values. We distinguish the pointer type in our system so that we can give generic typing rules for memory reads and writes.

The notation $\texttt{Rec}_s.\langle 0 : t_0;\ \dots\ ;\ n-1 : t_{n-1}\rangle$ denotes a very-dependent [7] record type with n mutable fields, each of whose types may depend on the runtime values of other fields. For simplicity, fields are labeled with their index in the record. The dependent type constructor "$\texttt{Rec}_s$" binds a variable s that can be thought of as the "self pointer" for the record. We use s to encode dependencies among the fields of the record: the special expression $s.i$ refers to the value stored in the ith word of the current record, where i is a constant. We say that a field type $C(d_1, \dots d_n)$ *refers to* field i iff at least one expression d_j is "$s.i$". A record type $\sigma = \texttt{Rec}_s.\langle 0 : t_0;\ \dots\ ;\ n-1 : t_{n-1}\rangle$ is *well-formed* if for all terms $s.j$ referring to a field, we have $0 \leq j < n$. In other words, dependencies must refer to fields that actually exist. We require that all types used in this framework be well-formed.

For example, a type policy may define the singleton type constructor Single(e), and then can define a dependent record containing two identical integers as

$$\texttt{Rec}_s.\langle 0 : \text{Int};\ 1 : \text{Single}(s.0)\rangle$$

If we define the type constructor "Array(len)" to be the type of a pointer to an array of Ints with length len, then a record containing an array pointer and the length of that array has the type

$$\texttt{Rec}_s.\langle 0 : \text{Array}(s.1);\ 1 : \text{Int}\rangle$$

Field types can even refer directly to the self pointer s. $\texttt{Rec}_s.\langle 0 : \text{Single}(s)\rangle$ is a one-word object that contains a pointer to itself. Circular dependencies are also allowed, so

$$\texttt{Rec}_s.\langle 0 : \text{Single}(s.1);\ 1 : \text{Single}(s.0)\rangle$$

is another valid definition for our record containing two identical integers.

We therefore have two kinds of memory locations in the language. *Dependent* fields have types that refer to the self pointer or other fields, or are referred to by the types of sibling fields. *Non-dependent* fields have types of the form C (or $C(c_1, \ldots c_n)$, where each c_i is a constant) that do not refer to, and are not referred to by, any other field. We must be careful when a dependent field is updated, to ensure that the dependencies are respected. However, we can modify non-dependent fields in place without additional checking.

We also support dependent function types, including function pointers. Checking dependent functions is very similar to checking that dependent records are used correctly, and we do not discuss them further here.

3.1 Type Checking

We describe here the process of typechecking assembly code when the start of each basic block has been annotated with an invariant, as is done in TAL [8]. For space reasons, we do not discuss in this paper our inference system for generating such invariants.

Figure 2 shows the simple MIPS-like assembly language that we will be type-checking. A basic block is a sequence of instructions whose entry is denoted by some label, and whose exit is a branch or a jump. Note that in this paper, we omit details relating to stack handling or the calling convention [9]. Our implementation uses the stack analysis engine written for the Open Verifier project [10].

We must track the memory state explicitly in order to reason about writes to dependent fields. "$\texttt{upd}(m, e_1, e_2)$" denotes the memory state that results from modifying memory state m by writing value e_2 at location e_1, while "$\texttt{sel}(m,\ e)$" is the result of reading address e in memory state m. We define "`ValidMem`" to be the type of a memory heap that is in a *consistent* state: one where all allocated locations contain a value that adheres to the type that the location was assigned when it was allocated. Consistency may be temporarily broken when we write a dependent field, since in general we will have to write to all of the fields in a dependent group before we can conclude that the group is consistent. But we will check that consistency holds at basic block boundaries.

instructions	$I ::= \texttt{mov}\ \mathrm{r}_{dest}, c \mid \texttt{mov}\ \mathrm{r}_{dest}, L \mid alu\ \ \mathrm{r}_{dest}, \mathrm{r}_{s1}, \mathrm{r}_{s2}$ $\mid \texttt{load}\ \mathrm{r}_{dest},\ \mathrm{r}_a \mid \texttt{store}\ \mathrm{r}_{src},\ \mathrm{r}_a$
arithmetic	$alu ::= \texttt{add} \mid \texttt{mult} \mid \texttt{xor} \mid \texttt{slt} \mid \ldots$
labels	L
jumps	$J ::= \texttt{beq}\ \mathrm{r}_c, L \mid \texttt{jump}\ L \mid \texttt{jr}\ \mathrm{r}$
basic blocks	$B ::= I, B \mid J$
functions	$F ::= \langle L_1 : B_1,\ \ldots,\ L_m : B_m \rangle$

Fig. 2. The target assembly language

states	$S ::= \langle \Delta, \Gamma, m \rangle$
register states	$\Delta ::= \mathrm{r}_1 = e_1,\ \ldots,\ \mathrm{r}_k = e_k$
type states	$\Gamma ::= v_1 \mapsto \tau_1,\ v_2 \mapsto \tau_2, \ldots$
memory states	$m ::= \texttt{upd}(m, e_1, e_2) \mid v$
abstract values	v
register types	$\tau ::= C(e_1, \ldots e_n) \mid \mathrm{Ptr}\ \sigma$
symbolic expressions	$e ::= c \mid v \mid L \mid \texttt{sel}(m,\ e) \mid e_1\ op\ e_2$
binary operations	$op ::= + \mid \times \mid \texttt{xor} \mid < \mid \ldots$

Fig. 3. The states of our symbolic execution algorithm for typechecking

Our typechecker performs symbolic evaluation on one basic block at a time, using abstract values v for any unknown values. As seen in Figure 3, a state in our checker is $\langle \Delta, \Gamma, m \rangle$, where Δ is a mapping from registers to symbolic expressions, Γ is a mapping from abstract values to types, and m is the current memory state. We could represent a checker state in which r_2 was known to equal $\mathrm{r}_1 + 1$ as $\langle \Delta_0, \Gamma_0, v_{mem0} \rangle$, where:

$$\Delta_0 = \{\mathrm{r}_1 = v;\ \mathrm{r}_2 = v + 1\}$$
$$\Gamma_0 = \{v \mapsto \mathrm{Int}; v_{mem0} \mapsto \texttt{ValidMem}\}$$

This state can be considered syntactic sugar for the following logical formula:

$$\exists v \in \mathrm{Int}\,.\,\exists v_{mem0} \in \texttt{ValidMem}\ .\ (\mathrm{r}_1 = v) \wedge (\mathrm{r}_2 = v + 1)$$

Typing Expressions. We give here the rules for assigning types to symbolic expressions. The judgment $\Gamma \vdash e : \tau$ means that expression e has type τ in context Γ. Most of the this work is done by the type policies through the functions `ConstType`, `IsSubtype`, and `ArithType`, while the framework maintains the types of abstract variables and handles the typing of memory operations.

$$\frac{}{\Gamma \vdash v : \Gamma(v)}\ [\text{Abstract}] \qquad \frac{\tau = \texttt{ConstType}(c)}{\Gamma \vdash c : \tau}\ [\text{Const}]$$

$$\frac{\Gamma \vdash e : \tau' \qquad \texttt{IsSubtype}(\tau',\ \tau)}{\Gamma \vdash e : \tau}\ [\text{Subsumption}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau = \texttt{ArithType}(\tau_1, op, \tau_2)}{\Gamma \vdash e_1 \; op \; e_2 : \tau} \text{[Arith]}$$

Type policies that do not care about arithmetic can say that $\texttt{ArithType}(\tau_1, op, \tau_2)$ is Int for all inputs, but policies such as CCured will derive a more precise type for some inputs to `ArithType`.

The final form of expression is a read from memory. When reading a dependent field with type $C(s.j)$, we must replace dependency $s.j$ with a symbolic expression that explicitly encodes the current value of the jth field. Consider a record that contains an array pointer and its length, and suppose we read the array field into r_1 and the length field into r_2: [1]

$$\Delta = \{\mathrm{r}_1 = \texttt{sel}(m_0,\ v);\ \mathrm{r}_2 = \texttt{sel}(m_0,\ v+1)\}$$
$$\Gamma = \{v \mapsto \text{Ptr } \texttt{Rec}_s.\langle 0 : \text{Array}(s.1);\ 1 : \text{Int}\rangle\}$$

The value in r_1 should have type "Array($\texttt{sel}(m_0,\ v+1)$)," to reflect the fact that the length of the array is located at address $v+1$ in memory state m_0. We can now use r_2 as the length of array r_1. Even if memory is later changed, for example by updating this record with a new array and different length, we will still be able to use r_2 as the length of r_1 since we remember that they were read from the same memory state m_0.

We generalize the above intuition into the following rule:

$$\frac{\begin{array}{l}\Gamma \vdash e : \text{Ptr } \texttt{Rec}_s.\langle 0 : t_0;\ \ldots\ ;\ n-1 : t_{n-1}\rangle \\ \tau = t_i[{}^{e}/_{s}][{}^{\texttt{sel}(m,\ e+0)}/_{s.0}]\cdots[{}^{\texttt{sel}(m,\ e+n-1)}/_{s.(n-1)}] \\ \Gamma \vdash m : \texttt{ValidMem}\end{array}}{\Gamma \vdash \texttt{sel}(m,\ e+i) : \tau} \text{[Read]}$$

The binding step $\tau = t_i[{}^{e}/_{s}][{}^{\texttt{sel}(m,\ e+0)}/_{s.0}]\cdots[{}^{\texttt{sel}(m,\ e+n)}/_{s.n}]$ will, for example, convert the field type Array($s.1$) from the previous example to the register type Array ($\texttt{sel}(m,\ v+1)$). The requirement $\Gamma \vdash m : \texttt{ValidMem}$ ensures that we are not in the middle of a dependent update.

Memory Updates. After writing a value to memory, we must see whether $\Gamma \vdash m : \texttt{ValidMem}$ for the resulting memory state m. If the `store` wrote to a dependent field, then other fields in the record may have to be updated as well in order for the record to be internally consistent once again. For simplicity, our framework requires that all the relevant dependent fields of a record be mutated in the same basic block, with no other intervening writes to the heap. However, it would not be hard to extend the type system to allow invalid memory states that span basic block boundaries.

The rule for stores is below. Starting from a consistent state m, a basic block can perform a series of writes to some object that starts at address e_a. The notation $\texttt{upd}(\cdot, e_a + c_i, e_i)$ represents the result of storing e_i into the object's ${c_i}^{th}$

[1] Throughout this paper we assume that memory is addressed by words, not bytes.

field; we check that each c_i is in bounds while typechecking the corresponding `store` statement. We ignore duplicate writes to the same field. Regardless of which fields have been overwritten, we can reestablish consistency for this object by checking whether *every* field $e_a + i$ in memory state m' has the type it should. First, we define a function that computes a canonical form for the result of a memory read using standard axioms for memory:

$$\text{Read}(m, e_a + i) = \begin{cases} e & \text{if } m = \texttt{upd}(m', e_a + i, e) \\ \text{Read}(m', e_a + i) & \text{if } m = \texttt{upd}(m', e_a + j, e) \text{ and } i \neq j \\ \texttt{sel}(v_{mem},\ e_a + i) & \text{if } m = v_{mem} \end{cases}$$

With this function we can write the axiom for validating a sequence of writes to the same record:

$$\frac{\begin{array}{l} m' = \texttt{upd}((\ldots\texttt{upd}(m, e_a + c_1, e_1)\ldots), e_a + c_j, e_j) \\ \Gamma \vdash e_a : \text{Ptr } \texttt{Rec}_s.\langle 0 : t_0;\ \ldots\ ;\ n-1 : t_{n-1}\rangle \\ \forall 0 \leq i < n\ .\ \Gamma \vdash \text{Read}(m', e_a + i) : \tau_i \\ \qquad \text{where } \tau_i = t_i[{}^{e_a}/_{s}][{}^{\text{Read}(m', e_a + 0)}/_{s.0}]\cdots[{}^{\text{Read}(m', e_a + n)}/_{s.n}] \\ \Gamma \vdash m : \texttt{ValidMem} \end{array}}{\Gamma \vdash m' : \texttt{ValidMem}} \text{[Update]}$$

For example, consider a record that contains an array reference, its length, and one other field of type Foo. Suppose r_2 contains an array pointer and that r_3 contains its length:

$$\begin{aligned} \Delta &= \{r_1 = v_{ptr};\ r_2 = v_2;\ r_3 = v_3\} \\ \Gamma &= \{v_{ptr} \mapsto \text{Ptr } \texttt{Rec}_s.\langle 0 : \text{Array}(s.1);\ 1 : \text{Int};\ 2 : \text{Foo}\rangle; \\ &\qquad v_2 \mapsto \text{Array}(v_3);\ v_3 \mapsto Int;\ v_{mem0} \mapsto \texttt{ValidMem}\} \end{aligned}$$

Now we update the memory state v_{mem0} writing v_2 at address r_1 and v_3 at address r_1+1, therefore mutating both the array and length fields of the record. These two store instructions produce the memory state

$$m' = \texttt{upd}(\texttt{upd}(v_{mem0}, v_{ptr}, v_2),\ v_{ptr} + 1,\ v_3)$$

The intermediate memory state $\texttt{upd}(v_{mem0}, v_{ptr}, v_2)$ is not consistent, and in general it must not be used for `load` instructions. But m' is consistent. Observe that we get

$$\begin{aligned} \text{Read}(m', v_{ptr} + 0) &= v_2 \\ \text{Read}(m', v_{ptr} + 1) &= v_3 \\ \text{Read}(m', v_{ptr} + 2) &= \texttt{sel}(v_{mem0},\ v_{ptr} + 2) \end{aligned}$$

Each of these three fields has the correct type. v_2 has type

$$\begin{aligned} \text{Array}(v_3) &= \text{Array}(\texttt{sel}(m',\ v_{ptr} + 1)) \\ &= \text{Array}(s.1)[{}^{\text{Read}(m', v_{ptr} + 1)}/_{s.1}] \end{aligned}$$

while v_3 has type Int. Location $v_{ptr} + 2$ was not modified, so we rely on the fact that "$\Gamma \vdash v_{mem0} : \texttt{ValidMem}$" holds to ensure that $\texttt{sel}(m',\ v_{ptr} + 2) = \texttt{sel}(v_{mem0},\ v_{ptr} + 2)$ has a value of type Foo.

Checking Basic Blocks. Now we can put these rules together to create a complete algorithm for typechecking a basic block according to the type policy.

The transition function for symbolic evaluation is straightforward. The effect of each instruction on a state $\langle \Delta, \Gamma, m \rangle$ is as follows:[2]

$$\begin{array}{l}
\langle \Delta, \Gamma, m \rangle \vdash \texttt{mov}\ \mathrm{r}_{dest}, c \Downarrow \langle \Delta[\mathrm{r}_{dest} \mapsto c],\ \Gamma,\ m \rangle \\
\langle \Delta, \Gamma, m \rangle \vdash \texttt{load}\ \mathrm{r}_{dest}, \mathrm{r}_a \Downarrow \langle \Delta[\mathrm{r}_{dest} \mapsto \texttt{sel}(m,\ \Delta(\mathrm{r}_a))],\ \Gamma,\ m \rangle \\
\langle \Delta, \Gamma, m \rangle \vdash \texttt{store}\ \mathrm{r}_{src}, \mathrm{r}_a \Downarrow \langle \Delta,\ \Gamma,\ \texttt{upd}(m, \Delta(\mathrm{r}_a), \Delta(\mathrm{r}_{src})) \rangle \\
\langle \Delta, \Gamma, m \rangle \vdash \texttt{add}\ \mathrm{r}_{dest}, \mathrm{r}_{s1}, \mathrm{r}_{s2} \Downarrow \langle \Delta[\mathrm{r}_{dest} \mapsto \Delta(\mathrm{r}_{s1}) + \Delta(\mathrm{r}_{s2})],\ \Gamma,\ m \rangle
\end{array}$$

The other ALU operations have rules similar to `add`. In addition to updating the state, we check that r_a contains a valid pointer in each `load` and `store` operation ($\Gamma \vdash \Delta(\mathrm{r}_a) : \mathrm{Ptr}\ \sigma$).

We assume that each basic block is annotated with an invariant in the form of a typechecker state $\langle \Delta_0, \Gamma_0, m_0 \rangle$, which we use as the initial state of our symbolic evaluation for the block. Evaluation then proceeds according to the transition rules above until we reach the end of the block. At this point we must check that the current state satisfies the invariant that is attached to the successor block(s).

One interesting case here is branches. The branch "`beq` r_1, L_j" at the end of block B_k means that control will jump to block B_j if $\mathrm{r}_1 = 0$, or fall through to B_{k+1} if $\mathrm{r}_1 \neq 0$. A branch may be a dynamic check of some fact that is interesting to the type policy. So each type policy can define an operation `Constrain` : $\langle \Delta, \Gamma, m \rangle \to e \to \langle \Delta, \Gamma, m \rangle$ that transforms a state to account for any relevant information in a branch condition. For example, suppose we have a state in which r_1 and r_2 hold the same possibly-NULL pointer to σ:

$$\begin{aligned}
\Delta_1 &= \{\mathrm{r}_1 = v_1, \mathrm{r}_2 = v_1\} \\
\Gamma_1 &= \{v_1 \mapsto \mathrm{MaybeNullPtr}\ \sigma\}
\end{aligned}$$

Then a typical type policy would define

$$\begin{array}{l}
\texttt{Constrain}(\langle \Delta_1, \Gamma_1, m_1 \rangle, \mathrm{r}_1 = 0) = \langle \{\mathrm{r}_1 = 0, \mathrm{r}_2 = 0\},\ \{\ \},\ m_1 \rangle \\
\texttt{Constrain}(\langle \Delta_1, \Gamma_1, m_1 \rangle, \mathrm{r}_1 \neq 0) = \langle \{\mathrm{r}_1 = v_1, \mathrm{r}_2 = v_1\},\ \{v_1 \mapsto \mathrm{Ptr}\ \sigma\},\ m_1 \rangle
\end{array}$$

We must check now that $\texttt{Constrain}(\langle \Delta_1, \Gamma_1, m_1 \rangle, \mathrm{r}_1 = 0)$ implies the invariant of B_j, and that $\texttt{Constrain}(\langle \Delta_1, \Gamma_1, m_1 \rangle, \mathrm{r}_1 \neq 0)$ implies the invariant of B_{k+1}.

4 Dependent Types for CCured

We have built a prototype checker and inference system for the CCured type system. CCured enforces type safety for legacy C code by classifying pointers according to their usage. Depending on a pointer's classification, or *kind*, CCured changes the pointer to a "fat" pointer structure that stores *metadata* such as array bounds and run-time type information. Figure 4 shows two such fat pointers that we support in our prototype implementation: RTTI pointers, which hold

[2] Changes to the program counter are omitted.

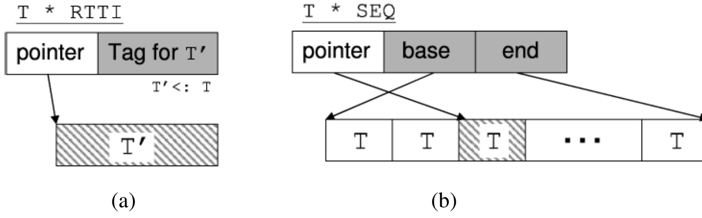


Fig. 4. Two "fat" pointer kinds used by CCured: (a) a pointer with run-time type information, and (b) a sequence pointer (array). The current targets of the pointers are shown with stripes, and the metadata added by the CCured code transformation is in grey.

Run-Time Type Information specifying the dynamic type of the object being pointed to, and Sequence pointers, which are used for arrays. The metadata is used to support run-time checks that CCured inserts when the pointer is dereferenced (for SEQ) or cast (for RTTI). When we want to update a pointer in memory, we may have to update all of the fields in the fat pointer.

4.1 RTTI Pointers

Figure 4(a) shows a two-word pointer that refers to a structure in memory and has a type tag specifying the run-time type of the object being pointed to. CCured stores the tag alongside the pointer instead of with the object itself for the sake of a less invasive transformation: the striped location could be in the middle of an array or a struct, and changing its representation to accommodate a type tag would mean transforming all accesses to the base type as well.

RTTI pointers are governed both by a static type (`T` in Figure 4) and the dynamic type specified by the tag ($\mathtt{T}'$), which must be a subtype of the static type. Before casting this pointer to a different type $\mathtt{T}''$, a program must check that the tag represents a subtype of $\mathtt{T}''$. CCured implements these checks using a global table that relates tag values to types.

The assembly-level definition of an RTTI pointer is given in Figure 5. The $\mathrm{Rtti}_\sigma(x)$ type constructor defines a possibly-NULL pointer that has the static type "pointer to σ" but that also has the type denoted by tag x. We use the function `typeof` here to encode the tags-to-types relation for each program.

Our prototype does not yet handle CCured's tagged unions or variable-argument functions, which require reasoning similar to RTTI pointers.

4.2 Sequence Pointers

CCured uses Sequence pointers to support arrays and pointer arithmetic in C. A Sequence pointer is a three-word fat pointer, as shown in Figure 4(b), consisting of the actual pointer and pointers to the two ends of the array.

The assembly-level encoding of these pointers is shown in Figure 5. The definition of Seq_σ directly follows the invariants that CCured maintains for its

$$\text{RTTI pointer to } \sigma = \texttt{Rec}_s.\langle 0 : \mathit{Rtti}_\sigma(s.1);\ 1 : \text{Int}\rangle$$
$$\text{Sequence pointer to } \sigma = \texttt{Rec}_s.\langle 0 : \text{Seq}_\sigma(s.1, s.2);\ 1 : \text{Int};\ 2 : \text{Int}\rangle$$

where

$$\begin{aligned}
\text{Rtti}_\sigma(t) &\triangleq \{p \mid (p = 0 \lor p \text{ isPtr } \sigma) \land (p = 0 \lor p \text{ isPtr } \texttt{typeof}(s.1))\\
\text{Seq}_\sigma(b, e) &\triangleq \{p \mid (b < e)\ \land\ (e - b) \bmod \texttt{sizeof}(\sigma) = 0\\
&\qquad \land (p - b) \bmod \texttt{sizeof}(\sigma) = 0\\
&\qquad \land \forall i.(b \leq (p{+}i{\cdot}\texttt{sizeof}(\sigma)) < e) \Rightarrow ((p + i \cdot \texttt{sizeof}(\sigma)) \text{ isPtr } \sigma)\}
\end{aligned}$$

Fig. 5. The meanings of the Rtti and Seq type constructors used by CCured. We use the set comprehension notation $\{x| \ldots\}$ to show the meanings of the types constructors, where "e isPtr σ" means that value e is a pointer to a record with type σ. The $<$ and $\leq$ operators used here are unsigned comparisons.

Sequence pointers: sequence is non-empty and both the end pointer and the actual pointer are aligned on multiples of the element size, although the pointer itself may be out of bounds. We can dereference a Seq_σ pointer p and treat it as an ordinary σ pointer if it is within its bounds b and e. Moreover, we can apply pointer arithmetic to this value, so long as the quantity being added is a multiple of the element size. If the new value is within the bounds, it too can be dereferenced.

To encode Sequence pointers in a type policy for our framework, we define a type constructor $\text{Seq}_\sigma(b, e)$ for each base type σ used by the program. We also define a constructor $\text{CheckedSeq}_\sigma(b, e)$ that represents a sequence pointer after a bounds check:

$$\begin{aligned}
&\text{CheckedSeq}_\sigma\ (b, e) \triangleq\\
&\qquad \{p \mid (b{<}e) \land (e - b) \bmod \texttt{sizeof}(\sigma){=}0 \land (p - b) \bmod \texttt{sizeof}(\sigma){=}0\\
&\qquad\quad \land \forall i.(b \leq (p{+}i{\cdot}\texttt{sizeof}(\sigma)) < e) \Rightarrow ((p + i \cdot \texttt{sizeof}(\sigma)) \text{ isPtr } \sigma)\\
&\qquad\quad \land b \leq p < e\}
\end{aligned}$$

CheckedSeq_σ has all of the properties of Seq_σ, meaning that we can do pointer arithmetic on it, as well as the property that the current value of the pointer is in bounds and can be dereferenced immediately. In the subtyping relationship used by `IsSubtype` and `TJoin`, $\text{CheckedSeq}_\sigma(e_b, e_e)$ is a subtype of both $\text{Seq}_\sigma(e_b, e_e)$ and Ptr σ.

Whenever our typechecker sees a bounds-checking branch instruction[3] for a value v_p that has type $\text{Seq}_\sigma(e_b, e_e)$, the `Constrain` operation refines the type of v into $\text{CheckedSeq}_\sigma(e_b, e_e)$. Now the value v_p can be dereferenced: the requirement in rules [Read] and [Update] that v_p have a pointer type is satisfied by the rule [Subsumption] and the fact `IsSubtype`($\text{CheckedSeq}_\sigma(e_b, e_e)$, Ptr σ).

[3] CCured checks both the lower and upper bounds of a sequence pointer in one branch instruction, by using the unsigned comparison $(\mathit{pointer} - \mathit{base}) < (\mathit{end} - \mathit{base})$.

For pointer arithmetic, we can define a type constructor MultipleOf(e) for the integers that are multiples of some value e, and we use the rules

$$\begin{array}{ll} \texttt{ArithType}(\text{Single}(c),\ \times,\ \text{Int}),\ \text{(where } c \text{ is a power of two}^4) & = \text{MultipleOf}(c) \\ \texttt{ArithType}(\text{Seq}_\sigma(e_b, e_e),\ +,\ \text{MultipleOf}(\texttt{sizeof}(\sigma))) & = \text{Seq}_\sigma(e_b, e_e) \\ \texttt{ArithType}(\text{CheckedSeq}_\sigma(e_b, e_e), +,\ \text{MultipleOf}(\texttt{sizeof}(\sigma))) & = \text{Seq}_\sigma(e_b, e_e) \end{array}$$

These rules let us assign the correct type $\text{Seq}_\sigma(e_b, e_e)$ to "$p + 4x$", where p has type $\text{CheckedSeq}_\sigma(e_b, e_e)$ and σ is 4 words long.

4.3 Other Features

Besides Rtti and Seq, our type system for CCured uses the basic type constructors you would expect for C code, such as MaybeNullPtr and Int. For each struct or base type defined in the source code, we create a record type σ.

Initialization. Allocation in C programs is done via calls to `malloc` or a related function. It is important to check that the newly-allocated data is initialized correctly. When allocating a record type that contains only non-dependent Ints, no initialization is needed since even garbage values are well-typed. But if the record contains pointer or dependent fields, those fields must be initialized to NULL. (By design, NULL is a valid value for every field type in CCured.)

Stack-allocated data. To support a common C programming idiom, we allow programs to take the address of locations on the stack and pass these pointers to other functions. Typically, this is done to achieve call-by-reference behavior. We require, however, that programs not store such pointers into heap locations or return them from functions. This restriction ensures that when the stack frame is deallocated, there are no dangling pointers into that stack frame. CCured's inference engine can tell us which arguments may be pointers to stack-allocated memory; the verifier needs simply to check that these pointers are not allowed to "escape" through the heap or a return value.

5 Implementation

We have implemented a prototype verifier for the output of CCured using the design in this paper. We use CCured to instrument C programs for type safety, and `gcc 3.3.3` to optimize and compile the code to x86 assembly. Our verifier uses abstract interpretation over the domain of symbolic expressions to infer register types and ensure that every instruction preserves memory safety. Our implementation can handle Sequence and RTTI pointers and their associated dependencies. We also implement pointers to stack-allocated data.

The CCured code transformer will generate annotations for each program that serve as a partial witness of the program's correctness, but these annotations need not be trusted. Incorrect annotations will result in failed verification

[4] When c is not a power of two, we need a branch instruction to check the result of the multiplication for overflow before assuming that the product is a multiple of c.

rather than unsoundness, just as incorrect type information in Java bytecode will result in failed typechecking. The annotations encode: (1) the type of every global variable; (2) the global table of RTTI tags; (3) for each function, the types of its arguments and return value, and the types and stack location of any local variables that will have their address taken; and (4) for every call to `malloc`, the type that will be applied to the resulting pointer (e.g. "`T*`" if the source instruction is "`T* var = (T*) malloc(`e`)`"). Annotations are expressed in inline assembly so that GCC will pass them from the instrumented source code down to the verifier. Only the annotations for `malloc` appear in the middle of a function, ensuring that this inline assembly will impose minimal constraints on the optimizer. Other annotation strategies would also be feasible.

These annotations give us all the information we need to know about the structure of the heap. All that remains is to infer types for registers and check each instruction for memory safety.

Considerable engineering work needs to be done before our verifier will be able to support all of the features of C. The prototype does not yet support variable-argument functions, tagged unions, floating point operations, or function pointers. We have not implemented any fat pointer kinds other than Sequence and RTTI, although most other kinds (such as "forward-sequence" and kinds that combine RTTI with bounds information) will be straightforward. We also do not support casts between Sequence pointer types that have different base types. Such casts are rare, and we may need CCured to annotate them so that they can be verified.

In order to facilitate joins, our abstract interpreter limits the form of symbolic expressions that are used for pointer arithmetic. Pointer offsets may be either constants or multiples of the base type size. This works well for one-dimensional arrays, but not for nested arrays. We are currently examining how to support more general indexing expressions without losing precision in our join algorithm.

We treat calls to `malloc` and other allocation functions specially, and deal with initialization as described in Section 4.3. CCured uses the Boehm-Demers-Weiser garbage collector [11], which we trust, so calling `free` has no effect.

5.1 Experiments

As an initial test, we used our prototype on the `go` program in the Spec95 benchmark suite. Of the Spec95 programs, we chose `go` because it makes extensive use of arrays while avoiding floating-point instructions, which our x86 parser does not yet handle. We used the `-O2` optimizer flag while compiling the program.

Of the 378 functions in the 29,321 LOC program, we can successfully verify 316 of them(84%). The most common reason for failure was that array indexing expressions of nested arrays are too complicated for our abstract domain. We directed CCured to flatten two-dimensional arrays into single-dimensional arrays, but in general there is no way to do this for arrays of structs that themselves contain arrays. Other failures were due to the unimplemented C features mentioned earlier. We are currently working to improve the implementation so that we can verify all of the Spec95 suite.

Verifying the program takes 194 seconds on a 2.4 GHz Pentium 4 with 1 GB of RAM. While testing our system, we discovered several soundness bugs in CCured: the instrumentation did not safely handle NULL return values from `malloc`, and CCured's optimizer incorrectly removed bounds checks based on the faulty assumption that two pointers couldn't alias. This experience shows the importance of independent verification of safety tools.

6 Related Work

Certified object code. There has been much work done to certify that binary code adheres to various safety properties. Colby et al. [12] survey several approaches, such as TAL and PCC, and describe the general problem of certifying mobile code, including how such certifications can be communicated to the end user.

Typed Assembly Language [8,4] is used as a compilation target for Popcorn, a subset of C. TAL includes many useful features, including flow-sensitive types for registers so that register types can change from one instruction to the next; typechecking that is done one basic block at a time; existential types; and support for stack-based compilation schemes [9]. But TAL does not support the dependent types that we need for CCured, and it assumes that assembly code is generated by a specially-written, type-preserving compiler.

Proof-Carrying Code [13,14] packages object code with a checkable proof of safety. The original implementations of PCC targeted specific type policies, such as Java's type system [14]. Recent projects such as LTT [15] and work by Shao et al. [16] seek a general type system for certified code that is not tied to any one source language. A low-level type system permits use of a wide variety of proofs and proof techniques, and it allows code from multiple source languages to be combined safely. But these two systems do not yet target imperative languages, making them impractical for the applications we are considering.

Producing checkable proofs is a goal for our type system as well. Our approach will follow work done by the Open Verifier group to design an extensible system for foundational verification [17]. Currently, our implementation uses the Open Verifier's code for checking that stacks and function calls are handled correctly.

Balakrishnan and Reps [18] present a system for analyzing memory accesses in x86 code. They do not require annotations from the compiler, but in exchange they trade off some precision and soundness.

Dependent types. The Xanadu language [19] provides an expressive dependent type system for an imperative, source-level language. Xanadu supports dependencies between different objects, which lets the language express more interesting properties about heap structures than ours can.

Xanadu can be compiled to DTAL, a dependently-typed assembly language [5]. DTAL focuses largely on array types and array-bound check elimination. Basic blocks are annotated with invariants to reduce the need for type inference, and a type-preserving compiler is used. DTAL does not support modification of dependently-typed locations in the heap.

Our restricted form of dependent types is similar to Hickey's very dependent function types [7]. Hickey encodes immutable records as functions from labels to values. By using very dependent types for these functions, one can impose dependencies among the object's fields. Hickey uses these types to formalize a theory of objects, including methods and inheritance. Our type system has a similar focus on dependencies among fields and function arguments, but in the context of a low-level imperative language with mutable structures.

Grossman [20] discusses the difficulty in supporting destructive updates in a language with existential types; this is the same difficulty that our system addresses for dependent types.

7 Discussion

We have described a dependently typed assembly language that supports destructive updates of dependent values that are stored in the heap. We can express in this framework the invariants enforced by CCured in the instrumented programs it outputs, and we can check statically that they are maintained. Our prototype verifier for CCured demonstrates that our approach can be used in practice.

Future work on this project will proceed in three main directions. First, we will apply our framework to type policies other than CCured. Already we have created an extension for Cqual [2], an interprocedural static analysis tool that infers type qualifiers for C programs and has been used to check several important security properties [21,22].

The second direction is to generalize our system of dependent types. Our types work well for dependencies between two local variables or two fields of the same object, but they cannot encode dependencies between two memory locations that are not stored in the same object. Removing this limitation will allow us to encode all or nearly all invariants of the source languages we are dealing with, including downcasts in object-oriented code and null-terminated strings.

The third direction of future work is to produce a proof of type safety for each program. Currently, the verifier and type policy are treated as part of the trusted computing base. Through the Open Verifier project [10], we plan to produce "foundational" proofs that can be checked by end users who would not need to trust our type inference or the implementation of the type policy.

References

1. Necula, G.C., Condit, J., Harren, M., McPeak, S., Weimer, W.: CCured: Type-safe retrofitting of legacy software. ACM Transactions on Programming Languages and Systems **27** (2005)
2. Foster, J.S., Terauchi, T., Aiken, A.: Flow-Sensitive Type Qualifiers. In: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, Berlin, Germany (2002) 1–12
3. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proc. 7th USENIX Security Conference, San Antonio, Texas (1998) 63–78

4. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. ACM Transactions on Programming Languages and Systems **21** (1999)
5. Xi, H., Harper, R.: Dependently Typed Assembly Language. In: The Sixth ACM SIGPLAN Int'l Conference on Functional Programming, Florence (2001) 169–180
6. Chang, B.Y.E., Chlipala, A., Necula, G., Schneck, R.: Type-based verification of assembly language for compiler debugging. In: The 2nd ACM SIGPLAN Workshop on Types in Language Design and Implementation. (2005) 91–102
7. Hickey, J.: Formal objects in type theory using very dependent types. In: Proceedings of the 3rd International Workshop on Foundations of Object-Oriented Languages. (1996)
8. Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., Zdancewic, S.: TALx86: A realistic typed assembly language. In: Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software. (1999) 25–35
9. Morrisett, G., Crary, K., Glew, N., Walker, D.: Stack-based typed assembly language. In: Proceedings of the Second International Workshop on Types in Compilation, Springer-Verlag (1998) 28–52
10. Schneck, R.R.: Extensible Untrusted Code Verification. PhD thesis, University of California, Berkeley (2004)
11. Boehm, H.J., Weiser, M.: Garbage collection in an uncooperative environment. Software—Practice and Experience (1988) 807–820
12. Colby, C., Crary, K., Harper, R., Lee, P., Pfenning, F.: Automated techniques for provably safe mobile code. Theor. Comput. Sci. **290** (2003) 1175–1199
13. Necula, G.C.: Proof-carrying code. In: The 24th Annual ACM Symposium on Principles of Programming Languages, ACM (1997) 106–119
14. Colby, C., Lee, P., Necula, G.C., Blau, F., Plesko, M., Cline, K.: A certifying compiler for java. In: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, ACM Press (2000) 95–107
15. Crary, K., Vanderwaart, J.C.: An expressive, scalable type theory for certified code. In: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, ACM Press (2002) 191–205
16. Shao, Z., Saha, B., Trifonov, V., Papaspyrou, N.: A type system for certified binaries. In: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (2002) 217–232
17. Chang, B.Y.E., Chlipala, A., Necula, G.C., Schneck, R.R.: The Open Verifier framework for foundational verifiers. In: The 2nd ACM SIGPLAN Workshop on Types in Language Design and Implementation. (2005) 1–12
18. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 binary executables. In: Proc. Compiler Construction (LNCS 2985), Springer Verlag (2004) 5–23
19. Xi, H.: Imperative programming with dependent types. In: Proceedings of 15th IEEE Symposium on Logic in Computer Science, Santa Barbara (2000) 375–387
20. Grossman, D.: Existential types for imperative languages. In: Proceedings of the 11th European Symposium on Programming Languages and Systems. (2002) 21–35
21. Shankar, U., Talwar, K., Foster, J.S., Wagner, D.: Detecting Format String Vulnerabilities with Type Qualifiers. In: Proceedings of the 10th Usenix Security Symposium, Washington, D.C. (2001)
22. Johnson, R., Wagner, D.: Finding user/kernel pointer bugs with type inference. In: Proceedings of the 13th USENIX Security Symposium. (2004)

The PER Model of Abstract Non-interference

Sebastian Hunt[1] and Isabella Mastroeni[2]

[1] Department of Computing, School of Informatics,
City University, London, UK
seb@soi.city.ac.uk

[2] Department of Computing and Information Sciences,
Kansas State University, Manhattan, Kansas, USA
isabellm@cis.ksu.edu

Abstract. In this paper, we study the relationship between two models of secure information flow: the PER model (which uses equivalence relations) and the abstract non-interference model (which uses upper closure operators). We embed the lattice of equivalence relations into the lattice of closures, re-interpreting abstract non-interference over the lattice of equivalence relations. For narrow abstract non-interference, we show that the new definition is equivalent to the original, whereas for abstract non-interference it is strictly less general. The relational presentation of abstract non-interference leads to a simplified construction of the most concrete harmless attacker. Moreover, the PER model of abstract non-interference allows us to derive unconstrained attacker models, which do not necessarily either observe all public information or ignore all private information. Finally, we show how abstract domain completeness can be used for enforcing the PER model of abstract non-interference.

Keywords: Information flow, non-interference, abstract interpretation, language-based security.

1 Introduction

An important task of language based security is to protect confidentiality of data manipulated by computational systems. Namely, it is important to guarantee that no information, about confidential/private data, can be caught by an external viewer. In the standard approach to the confidentiality problem, called *non-interference*, the characterization of attackers does not impose any observational or complexity restriction on the attackers' power. This means that the attackers are *all powerful*: they are modeled without any limitation in their quest to obtain confidential information. For this reason non-interference is an extremely restrictive policy. The problem of refining these security policies is considered as a major challenge in language-based information flow security [17]. Refining security policies means weakening standard non-interference, in such a way that it can be used in practice. Specifically, we need a weaker notion of non-interference where the power of the attacker (or external viewer) is bounded, and where intentional leakage of information is allowed.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 171–185, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Abstract non-interference is introduced [9] for modeling the *secrecy degree* of programs by means of abstract interpretation. In particular, it is possible to characterize the observational capability of the most powerful *harmless* attacker, that is, the most powerful attacker that cannot disclose any confidential information. Moreover, this model also allows one to characterize which aspects of private information can flow during the execution of a given program, when non-interference fails. These two complementary aspects of non-interference have been proved to be adjoint transformers of semantics in [10], where non-interference has been modeled as an abstract domain completeness problem.

In the PER model of secure information flow [18], a generalised notion of non-interference is obtained by using equivalence relations to model attackers. In this paper we show that, since equivalence relations can be viewed as particular types of closures called *partitioning* closures [16], the definitions of narrow and abstract non-interference from [9] can be re-interpreted by using equivalence relations only in place of arbitrary closures. For narrow abstract non-interference, we show that the new definition is equivalent to the original, whereas for abstract non-interference it is strictly less general. The difference lies in the fact that abstract non-interference depends on being able to distinguish properties of *sets* of values, such as intervals, congruences, etc, and this cannot be done with equivalence relations on the underlying set. We then show how the relational presentation of narrow abstract non-interference leads to a simplified construction of the most powerful harmless attacker. Moreover, the generalization of the PER model of secure information flow allows us to derive *unconstrained* attacker models, which do not necessarily either observe all public information or ignore all private information. Finally, we show how abstract domain completeness can be used for enforcing the PER model of abstract non-interference, proving that abstract non-interference corresponds to abstract domain completeness of the corresponding partitioning closures.

2 Mathematical Background

In this paper we use the standard framework of abstract interpretation [5,7] for modeling the observational capability of attackers. The idea is that, instead of observing the concrete semantics of programs, namely the values of public data, attackers can only observe *properties* of public data, namely an *abstract semantics* of the program. For this reason we model attackers by means of abstract domains. Abstract domains are used for denoting properties of concrete domains, since their mathematical structure guarantees, for each concrete element, the existence of the *best correct approximation* in the abstract domain. This is due to the fact that abstract domains are closed under the concrete greatest lower bound. The relation between abstract and concrete domains is formalized by Galois connections (GC). In GC-based abstract interpretation the concrete domain C and abstract domain A are often assumed to be complete lattices and are related by an abstraction map $\alpha : C \rightarrow A$ and concretization map $\gamma : A \rightarrow C$ form-

ing a GC $\langle C, \alpha, \gamma, A \rangle$ [5], i.e., for any $x \in C$ and $y \in A$: $\alpha(x) \leq_A y \Leftrightarrow x \leq_C \gamma(y)$. When α is surjective then the GC is said to be a Galois insertion (GI) and uniquely determines an abstract domain. Formally, the *lattice of abstract interpretations of* C is isomorphic to the lattice $uco(C)$ of all the upper closure operators on C [7]. An *upper closure operator* $\rho : C \to C$ on a poset C is monotone, idempotent, and extensive[1]. The dual notion of *lower closure operator* (lco) is a monotone, idempotent and reductive[2] map. Any closure operator is uniquely determined by the set of its fix points $\rho(C)$, which forms an abstract domain. If C is a complete lattice then $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \top, id \rangle$ is the lattice of upper closures, where $\top \stackrel{\text{def}}{=} \lambda x.\ \top$, $id \stackrel{\text{def}}{=} \lambda x.\ x$, and for every $\rho, \eta \in uco(C)$, $\{\rho_i\}_{i \in I} \subseteq uco(C)$ and $x \in C$: $\rho \sqsubseteq \eta$ iff $\eta(C) \subseteq \rho(C)$; $\sqcup_{i \in I} \rho_i = \bigcap_{i \in I} \rho_i$; and $\sqcap_{i \in I} \rho_i = \mathcal{M}(\bigcup_{i \in I} \rho_i)$, where $\mathcal{M}$ is the operation of closing a domain by concrete greatest lower bound, e.g., intersection on power domains. The *disjunctive completion* of an abstract domain $\rho \in uco(C)$ is the most abstract domain able to represent the concrete disjunction of its objects: $\curlyvee(\rho) = \sqcup\{\eta \in uco(C) | \eta \sqsubseteq \rho \text{ and } \eta \text{ is additive}\}$. ρ is disjunctive (or additive) iff $\curlyvee(\rho) = \rho$ (cf. [7]).

2.1 Equivalence Relations vs Closure Operators

In this section we review the relationships between equivalence relations and upper closures which are key to the development in the rest of the paper.

The lattice of equivalence relations. The equivalence relations on a set C form a lattice $\langle Eq(C), \sqsubseteq, \sqcap, \sqcup, Id_C, All_C \rangle$, where Id_C is the relation that distinguishes all the elements in C, All_C is the relation that cannot distinguish any element in C, and:

- $\mathtt{Q} \sqsubseteq \mathtt{R}$ iff $\mathtt{Q} \subseteq \mathtt{R}$ iff $x\,\mathtt{Q}\,y \;\Rightarrow\; x\,\mathtt{R}\,y$;
- $\mathtt{Q} \sqcap \mathtt{R} = \mathtt{Q} \cap \mathtt{R}$, i.e., $x\,\mathtt{Q} \sqcap \mathtt{R}\,y$ iff $x\,\mathtt{Q}\,y \;\wedge\; x\,\mathtt{R}\,y$;
- $\mathtt{Q} \sqcup \mathtt{R} = \mathbb{T}(\mathtt{Q} \cup \mathtt{R})$, where $x\,\mathtt{Q} \cup \mathtt{R}\,y$ iff $x\,\mathtt{Q}\,y \;\vee\; x\,\mathtt{R}\,y$.

Here $\mathbb{T}(\mathtt{S})$ is the transitive closure of the relation $\mathtt{S}$ (it is easily seen that both $\cup$ and $\mathbb{T}$ preserve symmetry and reflexivity).

Relating equivalence relations and upper closures. In this paper we will generally be concerned with relationships between equivalence relations on a set C and upper closure operators on the powerset $\wp(C)$. However, we start by observing the following strong correspondence between ucos (on any lattice) and their own kernels. (Recall that the kernel, $\mathtt{K}_f$, of a function $f : C \to D$, is the equivalence relation on C defined by $x\ \mathtt{K}_f\ y$ iff $f(x) = f(y)$.)

Lemma 1. *Let* $\eta, \rho \in uco(C)$*. Then* $\eta \sqsubseteq \rho$ *iff* $\mathtt{K}_\eta \sqsubseteq \mathtt{K}_\rho$*.*

[1] $\forall x \in C.\ x \leq_C \rho(x)$.
[2] $\forall x \in C.\ x \geq_C \rho(x)$.

Next, we recall that there exists an isomorphism between equivalence relations and a subclass of the upper closure operators [16]. In fact, this isomorphism arises from a Galois connection between $Eq(C)$ and $uco(\wp(C))$. For each equivalence relation on a set C, $\mathtt{R} \subseteq C \times C$, we can define an upper closure operator on $\wp(C)$, $Clo^{\mathtt{R}} \in uco(\wp(C))$, and vice versa, from each upper closure operator $\eta \in uco(\wp(C))$ we can define an equivalence relation $Rel^{\eta} \subseteq C \times C$.

Consider an upper closure operator $\eta \in uco(\wp(C))$. We define $Rel^{\eta} \subseteq C \times C$, as $\forall x, y \in C \,.\, x \; Rel^{\eta} \; y \;\Leftrightarrow\; \eta(\{x\}) = \eta(\{y\})$. Proving that Rel^{η} is an equivalence relation is immediate and doesn't depend on the fact that η is a uco, but only on the fact that it is a function.

Consider now an equivalence relation $\mathtt{R} \subseteq C \times C$. We define $Clo^{\mathtt{R}} \in uco(\wp(C))$ as follows: $\forall x \in C \,.\, Clo^{\mathtt{R}}(\{x\}) = [x]_{\mathtt{R}}$ and $\forall X \subseteq C \,.\, Clo^{\mathtt{R}}(X) = \bigcup_{x \in X}[x]_{\mathtt{R}}$. Thus $Clo^{\mathtt{R}}$ is obtained by disjunctive completion of the partition induced by $\mathtt{R}$. Proving that $Clo^{\mathtt{R}}$ is an upper closure operator is immediate. In particular idempotence derives directly from the fact that $\mathtt{R}$ is an equivalence relation.

In [16], $Clo^{\mathtt{R}}$ is identified as the most concrete uco η such that $\mathtt{R} = Rel^{\eta}$. More precisely:

Proposition 2. *Let C be any set.*

1. *The mappings defined above form a Galois connection between the lattice of equivalence relations on C and the lattice of upper closure operators on its powerset. That is, for all $\mathtt{R} \in Eq(C)$, $\eta \in uco(\wp(C))$: $Clo^{\mathtt{R}} \sqsubseteq \eta \Leftrightarrow \mathtt{R} \sqsubseteq Rel^{\eta}$.*
2. *For all $\mathtt{R} \in Eq(C)$, $Rel^{Clo^{\mathtt{R}}} = \mathtt{R}$.*

Corollary 3. *Let $\Pi(\eta)$ be defined by $\Pi(\eta) = Clo^{Rel^{\eta}}$.*

1. *$\Pi : uco(\wp(C)) \to uco(\wp(C))$ is a lower closure operator.*
2. *For all $\eta \in uco(\wp(C))$, $\Pi(\eta)$ is the (unique) most concrete closure that induces the same equivalence relation as η ($Rel^{\eta} = Rel^{\Pi(\eta)}$).*

The fix points of Π are termed the *partitioning* closures [16].

Proposition 4. *An upper closure operator $\eta \in uco(\wp(C))$ is partitioning, i.e., $\eta = \Pi(\eta)$, iff it is complemented, namely if $\forall X \in \eta.\ \overline{X} \stackrel{def}{=} C \smallsetminus X \in \eta$.*

Indeed, an upper closure operator η is always closed under glb (intersection in this context), therefore whenever it is closed also under complementation, we have that it is surely disjunctive, by De Morgan's laws. In the following we have an example of the partitioning closure associated with a partition.

Example 5. Consider the set $\Sigma = \{1, 2, 3, 4\}$ and one of its possible partitions $\pi = \{\{1\}, \{2, 3\}, \{4\}\}$, then the closure η with fix points $\{\varnothing, \{1\}, \{4\}, \{123\}, \Sigma\}$ induces exactly π as partition of states, but the most *concrete* closure that induces π is $Clo^{\pi} = \Pi(\eta) = \bigcurlyvee(\{\varnothing, \{1\}, \{2, 3\}, \{4\}\}, \Sigma)$, which is the closure on the right in Fig. 1.

On the closures we have the following characterizations. Note that, since Π is a lower closure operator on $uco(\wp(C))$, then $\sqcup$ in Eq coincides with $\sqcup$ in uco, whereas $Clo^{\mathtt{Q} \sqcap \mathtt{R}}$ can be strictly less than $Clo^{\mathtt{Q}} \sqcap Clo^{\mathtt{R}}$.

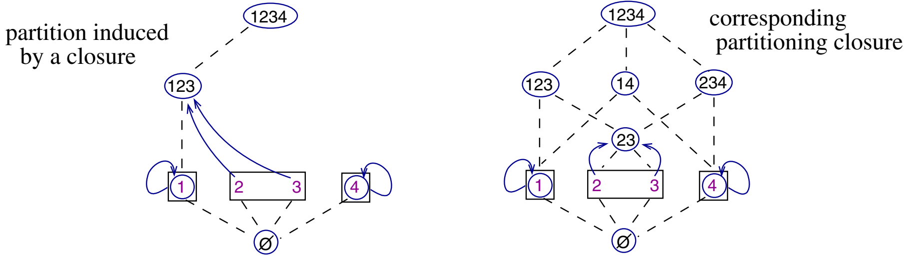


Fig. 1. A partitioning closure

Proposition 6. $\mathtt{Q} \sqsubseteq \mathtt{R}$ *iff* $Clo^{\mathtt{Q}} \sqsubseteq Clo^{\mathtt{R}}$, $\mathtt{Q} \sqcap \mathtt{R} = Rel^{Clo^{\mathtt{Q}} \sqcap Clo^{\mathtt{R}}}$ *and* $\mathtt{Q} \sqcup \mathtt{R} = Rel^{Clo^{\mathtt{Q}} \sqcup Clo^{\mathtt{R}}}$.

3 Information Flows in Language-Based Security

In the rest of this paper, confidential data are considered *private*, labeled with $\mathtt{H}$ (high level of secrecy), while all other data are public, labeled with $\mathtt{L}$ (low level of secrecy). Non-interference can be naturally expressed by using semantic models of program execution (this idea goes back to Cohen's work on *strong dependency* [3]). Non-interference for programs essentially means that *"a variation of confidential (high or private) input does not cause a variation of public (low) output"* [17]. When this happens, we say that the program has only *secure information flows* [1,3,8,13]. This situation has been modeled by considering the denotational (input/output) semantics $[\![P]\!]$ of the program P. Program states in Σ are functions (represented as tuples) mapping variables into the set of values $\mathbb{V}$. If $\mathtt{T} \in \{\mathtt{H}, \mathtt{L}\}$, $n = |\{x \in \mathit{Var}(P) | x : \mathtt{T}\}|$, and $v \in \mathbb{V}^n$, we abuse notation by denoting $v \in \mathbb{V}^{\mathtt{T}}$ the fact that v is a possible value for the variables with security type $\mathtt{T}$. Moreover, we assume that any input s, can be seen as a pair (h, l), where $s^{\mathtt{H}} = h$ is a value for private data and $s^{\mathtt{L}} = l$ is a value for public data. In this case, *(standard) non-interference* can be formulated as follows.

A program P is *secure* if $\forall$ input $s, t.\ s^{\mathtt{L}} = t^{\mathtt{L}} \ \Rightarrow\ ([\![P]\!](s))^{\mathtt{L}} = ([\![P]\!](t))^{\mathtt{L}}$

This definition has been formulated also as a *Partial Equivalence Relation* (PER) [18]. The standard methods for checking non-interference are based on security-type systems and data-flow/control-flow analysis. Type-based approaches are designed in such a way that well typed programs do not leak secrets. In a security-typed language, a type is inductively associated at compile time with program statements in such a way that any statement showing a potential flow disclosing secrets is rejected [19,21]. Similarly, data-flow/control-flow analysis techniques are devoted to statically discover flows of secret data into public variables [2,13,15,18]. All these approaches are characterized by the way they model attackers (or unauthorized users).

Table 1. Narrow and Abstract Non-Interference

Notion	Definition
$[\eta]P(\rho)$	if $\forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}, \forall l_1, l_2 \in \mathbb{V}^{\mathtt{L}} \,.\, \eta(\{l_1\}) = \eta(\{l_2\}) \;\Rightarrow\; \rho(\{[\![P]\!](h_1, l_1)^{\mathtt{L}}\}) = \rho(\{[\![P]\!](h_2, l_2)^{\mathtt{L}}\})$
$(\eta)P(\phi \rightsquigarrow\!\!\| \rho)$	if $\forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}, \forall l \in \mathbb{V}^{\mathtt{L}} \,.\, \rho([\![P]\!](\phi(\{h_1\}), \eta(\{l\}))^{\mathtt{L}}) = \rho([\![P]\!](\phi(\{h_2\}), \eta(\{l\}))^{\mathtt{L}})$

3.1 Abstract Non-interference: Attack Models

The notion of abstract non-interference [9] is introduced for modeling both weaker attack models, and declassification. The idea is that an attacker can observe only some properties, modeled as abstract interpretations of program semantics, of public concrete values. The *model of an attacker*, also called *attacker*, is therefore a pair of abstractions $\langle \eta, \rho \rangle$, with $\eta, \rho \in uco(\wp(\mathbb{V}^{\mathtt{L}}))$, representing what an observer can see about, respectively, the input and output of a program. The notion of *narrow (abstract) non-interference* (NNI), denoted $[\eta]P(\rho)$, is given in Table 1. It says that if the attacker is able to observe the property η of public input, and the property ρ of public output, then no information flow concerning the private input is observable from the public output. The problem with this notion is that it introduces *deceptive flows* [9], generated by different public output due to different public input with the same η property. Consider, for instance, the program $l := l * h^2$ and an observer who can observe only the parity of l on input and its sign on output. Intuitively, we may say that no information flows from h, since the sign of l after the assignment does not reveal anything about the value of h. However, $[Par]l := l * h^2(Sign)$ does *not* hold[3], since there is variation of the output's sign due to the existence of both negative and positive even numbers. In order to avoid deceptive flows we introduce a weaker notion of non-interference, which considers as public input the *set* of all the elements sharing the same property η. Hence, in the previous example, the observable output for l is the set of all the elements with the same parity, e.g., if $Par(l) = even$ then we check the sign of $\left\{\, l * h^2 \,\middle|\, l \text{ is even} \,\right\}$ which is always unknown, since an even number can be both positive and negative, while h^2 *does not* interfere with the final sign. Moreover, we consider also a property $\phi \in uco(\wp(\mathbb{V}^{\mathtt{H}}))$, modeling the private property that has not to be observed by the attacker $\langle \eta, \rho \rangle$. This notion, denoted $(\eta)P(\phi \rightsquigarrow\!\!\| \rho)$, is called *abstract non-interference* (ANI) and is defined in Table 1. So for example the property $(id)l := l * h^2(Sign \rightsquigarrow\!\!\| Sign)$ is satisfied, since the public result's sign do not depend on the private input sign, which is kept secret.

Note that $[id]P(id)$ models exactly (standard) non-interference. Moreover, we have that abstract non-interference is a weakening of both standard and narrow non-interference: $[id]P(id) \;\Rightarrow\; (\eta)P(\phi \rightsquigarrow\!\!\| \rho)$ and $[\eta]P(\rho) \;\Rightarrow\; (\eta)P(\phi \rightsquigarrow\!\!\| \rho)$, while standard non-interference is not stronger than the narrow version, due to deceptive flows. In [9], two methods are provided for deriving the most concrete output observation for a program, given the input one, for both NNI and ANI. In particular the idea is to collect in the same abstract object all the elements

[3] Here $Par \stackrel{\text{def}}{=} \{\top, ev, od, \bot\}$ and $Sign \stackrel{\text{def}}{=} \{\top, 0+, -, \bot\}$.

that, if distinguished, would generate a visible flow. These most concrete output observations, unable to get information from the program P observing η in input, are, respectively, denoted $[\eta][\![P]\!](id)$ and $(\eta)[\![P]\!](\phi \rightsquigarrow\!\!\!\!\| id)$, both in $uco(\wp(\mathbb{V}^{\mathtt{L}}))$. Hence, if for instance $P \stackrel{\text{def}}{=} l := |l| * Sign(h)$ (where $|\cdot|$ is the absolute value), we note that each value n has to be abstracted together with its opposite $-n$, in order not to generate visible flows, hence the most concrete harmless attacker can at most observe the absolute value Abs, i.e., $[Abs][\![P]\!](id) = Abs$.

3.2 PER Model

The semantic approach described above has also been equivalently formalized in [18], by using *partial equivalence relations (PER)* to model dependencies in programs. As we noted above, the problem of non-interference can be seen as absence of dependencies among data, where the meaning of dependency is given in [3]. The idea behind this characterization consists in interpreting security types as partial equivalence relations. In particular the type H is interpreted by using the equivalence relation All, and L by using the relation Id. The intuition is that All and Id model, respectively, that the user has no access to the high information and has full access to the low information. This perspective can simply be generalized to multilevel security problems.

In order to use this model in the security framework we need to combine equivalence relations on simple domains to construct new relations on more complex domains, in particular product spaces and function spaces. For the latter, it turns out to be natural to generalise slightly to consider *partial* equivalence relations, that is, relations which are symmetric and transitive but not necessarily reflexive. Let $Per(D)$ be the set of partial equivalence relations on D. Given $\mathtt{P} \in Per(D)$ and $\mathtt{Q} \in Per(E)$ we define $(\mathtt{P} \twoheadrightarrow \mathtt{Q}) \in Per(D \rightarrow E)$ and $(\mathtt{P} \times \mathtt{Q}) \in Per(D \times E)$ as follows:

1. $f\ (\mathtt{P} \twoheadrightarrow \mathtt{Q})\ g \ \Leftrightarrow\ \forall x, x' \in D\,.\, x\ \mathtt{P}\ x' \ \Rightarrow\ f(x)\ \mathtt{Q}\ g(x')$
2. $\langle x, y\rangle\ \mathtt{P} \times \mathtt{Q}\ \langle x', y'\rangle \ \Leftrightarrow\ x\ \mathtt{P}\ x' \ \wedge\ y\ \mathtt{Q}\ y'$.

In general, for $\mathtt{P} \in Per(D)$ and $x \in D$, we write $x : \mathtt{P}$ to mean $x\ \mathtt{P}\ x$. In particular, if $f\ (\mathtt{P} \twoheadrightarrow \mathtt{Q})\ f$, we write $f : \mathtt{P} \twoheadrightarrow \mathtt{Q}$. Note that $\mathtt{P} \twoheadrightarrow \mathtt{Q}$ will not, in general, be reflexive, even when P and Q are (for example, $All \twoheadrightarrow Id$ relates only functions which are equal and *constant*).

At this point, we can formalize security in this model.

Definition 7. *[18] A program P is said to be* secure *iff* $\forall s, t\,.\, \langle s^{\mathtt{H}}, s^{\mathtt{L}}\rangle\ All \times Id\ \langle t^{\mathtt{H}}, t^{\mathtt{L}}\rangle \ \Rightarrow\ [\![P]\!](s)\ All \times Id\ [\![P]\!](t)$*, or, more concisely:* $[\![P]\!] : All \times Id \twoheadrightarrow All \times Id$.

4 PER Model Versus Abstract Non-interference

The correspondence existing between ucos and equivalence relations suggests that we can define particular notions of abstract non-interference where the closures modeling properties are all partitioning, i.e., correspond exactly to equiv-

alence relations. As shown below, for NNI this specialisation makes essentially no difference, while for ANI it does involve a loss of generality.

First of all we introduce the natural generalization of the PER model provided in [18]. Given a program P and relations $\mathtt{Q}, \mathtt{W} \in Eq(\mathbb{V})$, we say that P is $\langle \mathtt{Q}, \mathtt{W} \rangle$-secure iff $[\![P]\!] : \mathtt{Q} \twoheadrightarrow \mathtt{W}$. Clearly, P is secure (Definition 7) just when it is $\langle All \times Id, All \times Id \rangle$-secure.

4.1 PER Model vs NNI

Proposition 8. *Let P be a deterministic program. Let $\eta, \rho \in uco(\wp(\mathbb{V}^{\mathtt{L}}))$. Then:*

1. $[\eta]P(\rho)$ *iff* $[\![P]\!] : All \times Rel^{\eta} \twoheadrightarrow All \times Rel^{\rho}$
2. $[\eta]P(\rho)$ *iff* $[\Pi(\eta)]P(\Pi(\rho))$

Proof. Part 1 is immediate from the definitions. Part 2 follows from part 1 by part 2 of Corollary 3. □

Since every equivalence relation $\mathtt{R}$ is represented exactly by the uco $Clo^{\mathtt{R}}$, this result shows that precisely the same class of NNI properties can be expressed using equivalence relations or partitioning closures as using arbitrary ucos. In particular, we may define NNI directly in terms of equivalence relations:

Definition 9. *Let P be a program. Let $\mathtt{R}, \mathtt{S} \in Eq(\mathbb{V}^{\mathtt{L}})$. Then P is said to be $\langle \mathtt{R}, \mathtt{S} \rangle$-NSecret, written $[\mathtt{R}]P(\mathtt{S})$, iff $[\![P]\!] : All \times \mathtt{R} \twoheadrightarrow All \times \mathtt{S}$.*

By Proposition 8, all NNI properties may be written in this form.

4.2 PER Model vs ANI

To compare the relative expressive power of the PER model and the general notion of abstract non-interference using arbitrary ucos, it is helpful to consider the extension of a relation on C to a relation on subsets of C. The basic construction is that used in defining Plotkin's powerdomain.

Definition 10. *Let $\mathtt{R}$ be a binary relation on a set C. Then the extension of $\mathtt{R}$ to $\wp(C)$ is the relation $\mathcal{P}[\mathtt{R}] \subseteq \wp(C) \times \wp(C)$ such that $X \; \mathcal{P}[\mathtt{R}] \; Y$ iff*

$$\forall x \in X.\, \exists y \in Y \,.\, x \;\mathtt{R}\; y \;\; \textit{and} \;\; \forall y \in Y.\, \exists x \in X \,.\, x \;\mathtt{R}\; y$$

For a partitioning closure, the extension of its corresponding equivalence relation from C to $\wp(C)$ has a particularly simple characterisation:

Proposition 11. *Let C be any set and let $\eta \in uco(\wp(C))$ be partitioning. Then $\mathcal{P}[Rel^{\eta}] = \mathtt{K}_{\eta}$, that is: $X \; \mathcal{P}[Rel^{\eta}] \; Y \;\Leftrightarrow\; \eta(X) = \eta(Y)$.*

Corollary 12. *Let $\eta, \phi \in uco(\wp(\mathbb{V}^{\mathtt{L}}))$ and let $\rho \in uco(\wp(\mathbb{V}^{\mathtt{H}}))$. If ρ is partitioning, then $(\eta)P(\phi \rightsquigarrow\!\!\! \| \rho)$ iff*

$$\forall X_1, X_2 \in \mathbb{V}^{\mathtt{H}}/Rel^{\phi}, \forall Y \in \mathbb{V}^{\mathtt{L}}/Rel^{\eta} \,.\, [\![P]\!](X_1, Y) \; \mathcal{P}[All \times Rel^{\rho}] \; [\![P]\!](X_2, Y)$$

The following proposition shows that, in contrast to NNI, there are ANI properties which cannot be expressed using the partitioning closures alone.

Proposition 13. *Let P be a program, let $\eta, \phi \in uco(\wp(\mathbb{V}^{\mathtt{L}}))$ and $\rho \in uco(\wp(\mathbb{V}^{\mathtt{H}}))$. Then $(\Pi(\eta))P(\Pi(\phi) \rightsquigarrow\!\!\| \Pi(\rho)) \Rightarrow (\eta)P(\phi \rightsquigarrow\!\!\| \rho)$ but, in general, the reverse implication does not hold.*

The following example shows where the difference between the two notions lies.

Example 14. Consider the following program fragment:

$$P \stackrel{\text{def}}{=} \textbf{if } h = 0 \textbf{ then } l := l \; mod \; 6 + 2; \textbf{else if } l < 0 \textbf{ then } l := 2 \textbf{ else } l := 7;$$

with security typing $h : \mathtt{H}$, $l : \mathtt{L}$. Consider $\eta \stackrel{\text{def}}{=} \{\top, 2\mathbb{Z}, 2\mathbb{Z}+1, \bot\}$ for parity, $\phi = \{\top, 0+, -, \bot\}$ for sign, and $\rho \stackrel{\text{def}}{=} Int$ of intervals [5], in $uco(\wp(\mathbb{Z}))$. Note that, since each integer number is in particular an interval, we have that $\Pi(Int) = id$, distinguishing all the integer values, while $\Pi(\eta) = \eta$ and $\Pi(\phi) = \phi$. Let us see what happens in abstract non-interference. Consider $\eta(l) = 2\mathbb{Z}$, then if $\phi(h) = 0+$ we have that $\rho([\![P]\!](\phi(h), \eta(l))^{\mathtt{L}}) = \rho(\{2,4,6,7\}) = [2,7]$. While, if $\phi(h) = -$, then we have $\rho([\![P]\!](\phi(h), \eta(l))^{\mathtt{L}}) = \rho(\{2,7\}) = [2,7]$. On the other hand, if $\eta(l) = 2\mathbb{Z}+1$ and $\phi(h) = 0+$, then $\rho([\![P]\!](\phi(h), \eta(l))^{\mathtt{L}}) = \rho(\{2,3,5,7\}) = [2,7]$, and when $\phi(h) = -$ we have $\rho([\![P]\!](\phi(h), \eta(l))^{\mathtt{L}}) = \rho(\{2,7\}) = [2,7]$. So $(\eta)P(\phi \rightsquigarrow\!\!\| \rho)$ holds. Consider now $\Pi(\rho) = id$. It is clear that if we substitute above ρ with id, then we have that $(\Pi(\eta))P(\Pi(\phi) \rightsquigarrow\!\!\| \Pi(\rho))$ does not hold. □

Hence, ANI with ucos is a more precise notion whenever we have to deal with sets of values, instead of with singletons. This may be particularly useful, for example, for non-deterministic systems, where the denotational semantics returns a *set* of states as output.

5 Deriving Attacker Models by Abstract Interpretation

In this section we consider the PER model of NNI and use it to derive simple, constructive characterisations of various classes of attacker considered in [9]. For example, suppose given a class of attackers whose power to observe low security inputs is given by $\mathtt{R}$: for a given program P, what is the most powerful attacker in the class (with respect to observation of low security outputs), for which P is secure? There are two cases of principal interest:

1. *Most powerful attacker*: given $\mathtt{R} \in Eq(\mathbb{V}^{\mathtt{L}})$, is there a smallest $\mathtt{S} \in Eq(\mathbb{V}^{\mathtt{L}})$ such that $[\mathtt{R}]P(\mathtt{S})$? Or, given $\mathtt{S} \in Eq(\mathbb{V}^{\mathtt{L}})$, is there a greatest $\mathtt{R} \in Eq(\mathbb{V}^{\mathtt{L}})$ such that $[\mathtt{R}]P(\mathtt{S})$?
2. *Fix point (canonical) attacker*: is there a smallest $\mathtt{R}$ such that $[\mathtt{R}]P(\mathtt{R})$?

The particular interest of fix point attackers is that, in many situations, the power of the attacker to observe low security data may be independent of the data's rôle as input or output.

5.1 Deriving Unconstrained Attackers

In this section, given a semantics f and an input [output] equivalence relation $\mathtt{R}$ [$\mathtt{S}$], we show how we can derive the most concrete [abstract] output [input] relation $\mathtt{S}$ [$\mathtt{R}$] that makes the program satisfy $f : \mathtt{R} \twoheadrightarrow \mathtt{S}$. Consider an arbitrary function $f : A \to B$ between sets. As is well known, any such f lifts to an adjunction between $\wp(A)$ and $\wp(B)$, in the form of f's direct and inverse image mappings. It turns out that f can be lifted to an adjunction $\langle Eq(A), \widehat{f}, \widehat{f}^{-1}, Eq(B)\rangle$ between lattices of equivalence relations in a similar way. In this section we detail the construction of $\widehat{f}$ and $\widehat{f}^{-1}$, and we go onto show how they are used to derive attackers.

Given an output relation $\mathtt{S}$ it is always possible to find a good candidate for input relation $\mathtt{R}$, essentially by simply imposing the condition $f : \mathtt{R} \twoheadrightarrow \mathtt{S}$. In other words we can always define the equivalence relation $\widehat{f}^{-1}(\mathtt{S})$ in the following way:

$$x\ \widehat{f}^{-1}(\mathtt{S})\ y \text{ iff } f(x)\ \mathtt{S}\ f(y) \tag{1}$$

This is the key definition in [14] and is also exactly the idea used in [22] on the trace semantics, namely we collect together all the elements whose semantics are equivalent in the output observation.[4]

Lemma 15. *$\widehat{f}^{-1}(\mathtt{S})$ is an equivalence relation and $f : \mathtt{R} \twoheadrightarrow \mathtt{S} \Leftrightarrow \mathtt{R} \sqsubseteq \widehat{f}^{-1}(\mathtt{S})$.*

Note that for each $\mathtt{S}$ we have $\widehat{f}^{-1}(\mathtt{S}) \sqsupseteq \mathtt{K}_f$. This means that the input relation has, at least, to identify all the elements with the same image under f. This observation makes the definition of $\widehat{f}$ a bit more complicated. Indeed, given $\mathtt{R}$, we would like to find the best relation $\mathtt{S}$ which satisfies $f : \mathtt{R} \twoheadrightarrow \mathtt{S}$. A naive construction leads to the function $\widetilde{f} : Rel(C) \to Rel(C)$, as follows:

$$y\ \widetilde{f}(\mathtt{R})\ y' \text{ iff } (\exists x, x' \,.\, x\ \mathtt{R}\ x' \text{ and } f(x) = y,\ f(x') = y' \ \vee\ y = y')$$

Note that the disjunct $y = y'$ guarantees that the relation is reflexive. However, $\widetilde{f}(\mathtt{R})$ may fail to be transitive, as we can see in the following example.

Example 16. Consider a domain $C = \{1, 2, 3, 4, 5, 6\}$ and a function f such that $f(1) = 1$, $f(3) = f(4) = 2$, $f(2) = f(6) = 5$ and $f(5) = 3$, and suppose that $\mathtt{R} = \{[1, 3], [2, 4], [5, 6]\}$, then we would have $1\widetilde{f}(\mathtt{R})2$, $2\widetilde{f}(\mathtt{R})5$ and $5\widetilde{f}(\mathtt{R})3$, but for example $1\neg\widetilde{f}(\mathtt{R})3$.

The problem is that f is not injective ($\mathtt{K}_f \neq Id$) and therefore, in the example the fact that $f(3) = f(4)$ while $\mathtt{R}$ distinguishes 3 from 4, creates the problems.

Proposition 17. *Consider $f : C \to C$ and $\mathtt{R} \in Eq(C)$. If $\mathtt{K}_f \sqsubseteq \mathtt{R}$, then $\widetilde{f}(\mathtt{R})$ is an equivalence relation, if $\mathtt{K}_f = \mathtt{R}$, then $\widetilde{f}(\mathtt{R}) = Id$.*

We would like to modify $\widetilde{f}$ in order to guarantee that $\widetilde{f}(\mathtt{R})$ is always an equivalence relation. For this reason we prove the following result.

[4] This transformation corresponds to the *quotient* of the concrete semantic domain with respect to the property $Clo^{\mathtt{S}}$ [4].

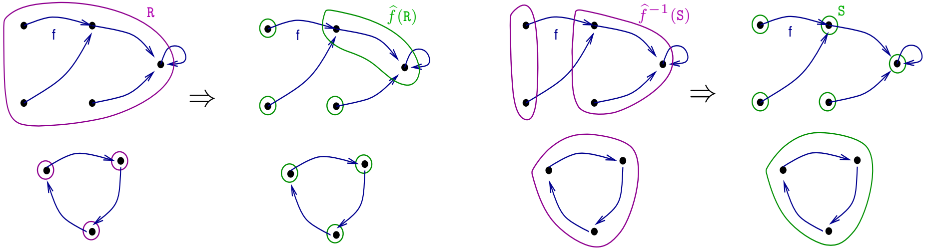


Fig. 2. Example of application of $\widehat{f}$ and of $\widehat{f}^{-1}$

Proposition 18. *Let $f : A \to B$. Then $\widehat{f}^{-1} : Eq(B) \to Eq(A)$ is co-additive.*

This means that $\widehat{f}^{-1}$ is the right adjoint of a Galois connection. Thus we can define the following function, which is its left adjoint [5]:

$$\widehat{f}(\mathtt{R}) \stackrel{\text{def}}{=} \bigsqcap \left\{ \mathtt{Q} \,\middle|\, \mathtt{R} \sqsubseteq \widehat{f}^{-1}(\mathtt{Q}) \right\} \tag{2}$$

The co-additivity of $\widehat{f}^{-1}$ guarantees that the element uniquely exists. We manipulate this set obtaining that $\widehat{f}(\mathtt{R}) = \bigsqcap \{ \mathtt{Q} \,|\, x \,\mathtt{R}\, y \;\Rightarrow\; f(x) \,\mathtt{Q}\, f(y) \}$.

Theorem 19. $\widehat{f}(\mathtt{R}) = \widetilde{f}(\mathtt{R} \sqcup \mathtt{K}_f) = \mathbb{T}(\widetilde{f}(\mathtt{R}))$.

This means that, when $\mathtt{R} \sqsupseteq \mathtt{K}_f$, then $\widetilde{f}(\mathtt{R}) = \widehat{f}(\mathtt{R})$.

By construction, the following result is straightforward:

Proposition 20. *$\langle Eq(A), \widehat{f}, \widehat{f}^{-1}, Eq(B)\rangle$ is a Galois connection. That is, for all $\mathtt{R} \in Eq(A), \mathtt{S} \in Eq(B)$: $\widehat{f}(\mathtt{R}) \sqsubseteq \mathtt{S} \Leftrightarrow \mathtt{R} \sqsubseteq \widehat{f}^{-1}(\mathtt{S})$.*

Combining Proposition 20 with Lemma 15, gives:

Theorem 21. $f : \mathtt{R} \twoheadrightarrow \mathtt{S} \;\Leftrightarrow\; \widehat{f}(\mathtt{R}) \sqsubseteq \mathtt{S} \;\Leftrightarrow\; \mathtt{R} \sqsubseteq \widehat{f}^{-1}(\mathtt{S})$.

This result shows which is the rôle of the two operators $\widehat{f}$ and $\widehat{f}^{-1}$ in the whole construction. Indeed, by Theorem 21 we have that f satisfies non-interference, namely $f : \mathtt{R} \twoheadrightarrow \mathtt{S}$, iff $\widehat{f}(\mathtt{R}) \sqsubseteq \mathtt{S}$. This means that $\widehat{f}$ characterizes exactly the *most concrete output relation* that guarantees non-interference for f, fixed the input relation. By the adjunction relation we can also say that $f : \mathtt{R} \twoheadrightarrow \mathtt{S}$ iff $\mathtt{R} \sqsubseteq \widehat{f}^{-1}(\mathtt{S})$. Thus $\widehat{f}^{-1}$ characterizes the *most abstract input relation* that guarantees non-interference for f, fixed the output relation. Indeed, as expected, we can always abstract the output observation and we can always concretize the input one. Note that [9] *misses* exactly a construction of the input observation that makes a program secure, given the output one, while this is possible in this context since we are considering equivalence relations. An example is provided in Fig. 2.

5.2 Fix Point Attackers

In this section we look for the characterisation of attackers that observe the same property both in input and in output. The idea is to consider the fix points of

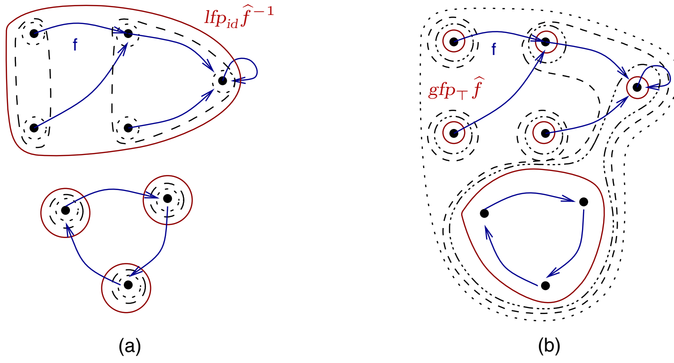


Fig. 3. Examples of fix points

the unconstrained attackers derived above. Unfortunately, the most concrete and the most abstract non trivial (different from top and identity) attacker models do not exist as can be also verified in Fig. 3, therefore we can use the fix point iteration simply as a possible systematic construction of canonical attackers.

Fix point of $\widehat{f}^{-1}$. Note that $\widehat{f}^{-1}(\top) = \top$, this means that the interesting case, if it exists, is the least fix point of $\widehat{f}^{-1}$ starting from *Id*. We know that $\widehat{f}^{-1}$ is monotone (Prop. 20), therefore the least fix point exists and can be obtained as the limit of the iterative application of $\widehat{f}^{-1}$ starting from *Id*, the bottom of the lattice of relations [6,20].

Fix point of $\widehat{f}$. Note that $\widehat{f}(Id) = Id$, this means that we can find, if it exists, only the greatest fix point of $\widehat{f}$ starting from $\top$. We know that $\widehat{f}$ is monotone (Prop. 20), therefore the greatest fix point exists and can be obtained as the limit of the iterative application of $\widehat{f}$ starting from the element $\top$ of the lattice of relations [6,20].

5.3 Deriving Contrained Attackers

In this section, we consider attackers which are unable to observe private data, and which can only observe properties of public data. In this way we derive attackers for abstract non-interference [9], where the attackers are modeled by equivalence relations instead of by closure operators.

Most Powerful Attackers. We can use $\widehat{f}$ to construct the most powerful attacker. Firstly, note that it follows directly from the definitions that $[\mathtt{R}]P(\mathtt{S})$ iff $\pi_2 \circ [\![P]\!] : All \times \mathtt{R} \twoheadrightarrow \mathtt{S}$[5]. The following result is then a straightforward consequence of Theorem 21:

Proposition 22. *Let P be a program and let* $\mathtt{R} \in Eq(\mathbb{V}^{\mathtt{L}})$. *Then the smallest* $\mathtt{S}$ *such that* $[\mathtt{R}]P(\mathtt{S})$ *is* $\widehat{f}(All \times \mathtt{R})$, *where* $f = \pi_2 \circ [\![P]\!]$.

[5] Here $\pi_2(\langle a, b \rangle) = b$ is the projection on the second component of a pair.

Fix Point Attackers. We wish to construct the smallest $\mathtt{R}$ such that:

$$[\![P]\!] : All \times \mathtt{R} \twoheadrightarrow All \times \mathtt{R} \tag{3}$$

Let $F_P(\mathtt{R}) \stackrel{\text{def}}{=} \widehat{f}(All \times \mathtt{R})$, where $f = \pi_2 \circ [\![P]\!]$. Then, using Theorem 21, it is easily verified that (3) holds iff $F_P(\mathtt{R}) \sqsubseteq \mathtt{R}$. Thus the solutions to (3) are just the post-fix points of F_P. Since F_P is clearly monotone on $Eq(\mathbb{V}^{\mathtt{L}})$, Tarski's fix point theorem gives:

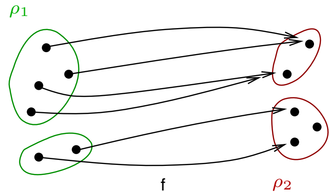
Proposition 23. *Let P be a program and let $F_P : Eq(\mathbb{V}^{\mathtt{L}}) \to Eq(\mathbb{V}^{\mathtt{L}})$ be defined as above. Then the smallest $\mathtt{R}$ such that $[\mathtt{R}]P(\mathtt{R})$ is lfp F_P.*

Note that this construction corresponds exactly to the characterization given in [9] for arbitrary closures. Indeed here we collect elements, in the new relation $\widehat{f}(\mathtt{R})$, iff they are images by f of elements that are in the input relation. In [9] the elements are collected, for obtaining the resulting closure, when they are images, under f of inputs that differ only in the private information (which is the input relation in ANI).

5.4 Non-interference and Completeness

In [10] it is proved that (abstract) non-interference can be modeled as a problem of completeness in the standard framework of abstract interpretation. Since partitions are particular closure operators, we can use completeness also for the PER model of abstract non-interference. We would like to understand how completeness can be helpful in order to obtain non-interference. First of all let us consider a new characterization of completeness.

Theorem 24. *Given $\rho_1, \rho_2 \in uco(\wp(C))$, and $f : C \to C$, then $\langle \rho_1, \rho_2 \rangle$ is complete for f, i.e., $\rho_2 \circ f \circ \rho_1 = \rho_2 \circ f$ iff $\forall X \in \rho_1 . \exists Y \in \rho_2$ such that $\forall z . (\rho_1(z) = X \Rightarrow \rho_2(f(z)) = Y)$.*



At this point let us define completeness of equivalence relations in terms of completeness of the corresponding closure operators. Let $\mathtt{R}, \mathtt{S} \in Eq(C)$, and f a map on C: $\mathtt{S} \circ f \circ \mathtt{R} = \mathtt{S} \circ f$ iff $Clo^{\mathtt{S}} \circ f \circ Clo^{\mathtt{R}} = Clo^{\mathtt{S}} \circ f$.

Corollary 25. *$f : \mathtt{R} \twoheadrightarrow \mathtt{S}$ iff $\mathtt{S} \circ f \circ \mathtt{R} = \mathtt{S} \circ f$.*

(It is interesting to note that precisely this relationship was used in [12] to establish a correspondence between PER-based and projection-based program analyses. It holds generally for idempotent maps and their kernels.)

This means that we can use the constructive method given in [11] for making abstract domains complete. Clearly the result of this transformation need not be a partitioning closure, hence we have then to derive the partition associated with the complete domain. In this way we obtain a method for making equivalence relations complete.

6 Conclusion

In this paper we define abstract non-interference in terms of the PER model. In particular, we consider equivalence relations instead of arbitrary abstract domains. We show that the notion does not change for narrow non-interference, while it becomes less general when we consider abstract non-interference. And it is possible to show that, even if we lift PERs to sets then we cannot reach the generality of uco since lifted PERs correspond only to additive closures. The use of equivalence relations allows us to simplify the characterization of the most powerful harmless attacker. Moreover we can also derive distinguished attackers for the generic PER model of security ($\langle \mathtt{Q}, \mathtt{W} \rangle$-security, Sect. 4).

Finally, we show that the PER model of abstract non-interference can be rewritten as an abstract domain completeness problem. This result is interesting for us since it suggests how we may approach the problem of making partitioning closures complete, similarly to what is done in [11]. Such a result could be useful also in other fields of computer science, such as completeness in model checking [16]. In this paper we only provide the relation-based construction of the most powerful harmless attacker for the narrow case, which is the straightforward generalization of the PER model [18]. It could be interesting to investigate if the restriction to partitioning closures simplifies also the characterization of the harmless attacker for abstract non-interference.

Acknowledgments

We would like to thank Roberto Giacobazzi for his insightful comments and contribution of ideas, and the anonymous referees for their helpful suggestions for improvement.

References

1. D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE Corp. Badford, MA, 1973.
2. D. Clark, C. Hankin, and S. Hunt. Information flow for algol-like languages. *Computer Languages*, 28(1):3–28, 2002.
3. E. S. Cohen. Information transmission in sequential programs. *Foundations of Secure Computation*, pages 297–335, 1978.
4. A. Cortesi, G. Filé, and W. Winsborough. The quotient of an abstract interpretation. *Theor. Comput. Sci.*, 202(1-2):163–192, 1998.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of Conf. Record of the 4th ACM Symp. on Principles of Programming Languages* (*POPL '77*), pages 238–252. ACM Press, New York, 1977.
6. P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific J. Math.*, 82(1):43–57, 1979.
7. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of Conf. Record of the 6th ACM Symp. on Principles of Programming Languages* (*POPL '79*), pages 269–282. ACM Press, New York, 1979.

8. D. E. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
9. R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*, pages 186–197. ACM-Press, NY, 2004.
10. R. Giacobazzi and I. Mastroeni. Adjoining declassification and attack models by abstract interpretation. In *Proc. of the European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 295–310. Springer-Verlag, 2005.
11. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. of the ACM.*, 47(2):361–416, 2000.
12. Sebastian Hunt. Pers generalise projections for strictness analysis (extended abstract). In *Proc. 1990 Glasgow Workshop on Functional Programming*, Workshops in Computing, Ullapool, 1991. Springer-Verlag.
13. R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37:113–138, 2000.
14. J. Landauer and T. Redmond. A lattice of information. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 65–70. IEEE Computer Society Press, 1993.
15. P. Laud. Semantics and program analysis of computationally secure information flow. In *In Programming Languages and Systems, 10th European Symp. On Programming, ESOP*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag, 2001.
16. F. Ranzato and F. Tapparo. Strong preservation as completeness in abstract interpretation. In D. Schmidt, editor, *Proc. of the 13th European Symposium on Programming (ESOP'04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 18–32. Springer-Verlag, 2004.
17. A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE J. on selected ares in communications*, 21(1):5–19, 2003.
18. A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.
19. C. Skalka and S. Smith. Static enforcement of security with types. In *ICFP'00*, pages 254–267. ACM Press, New York, 2000.
20. A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–310, 1955.
21. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.
22. S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 15–23. IEEE Computer Society Press, 2001.

A Relational Abstraction for Functions

B. Jeannet[1], D. Gopan[2], and T. Reps[2]

[1] IRISA
Bertrand.Jeannet@irisa.fr
[2] Comp. Sci. Dept., Univ. of Wisconsin
{gopan, reps}@cs.wisc.edu

Abstract. This paper concerns the abstraction of sets of functions for use in abstract interpretation. The paper gives an overview of existing methods, which are illustrated with applications to shape analysis, and formalizes a new family of *relational* abstract domains that allows sets of functions to be abstracted more precisely than with known approaches, while being still machine-representable.

1 Introduction

A major strength of abstract interpretation is the ability create complex abstract domains from simpler ones [2]. In particular, (*i*) Galois connections can be composed, which allows a complex abstraction to be described as the composition of simpler ones, offering the ability to identify clearly the different kind of approximations that take place; (*ii*) given two abstractions for sets of elements, $\wp(D_i)$, $i = 1, 2$, there exist techniques for abstracting functions of signature $D_1 \rightarrow D_2$ [4].

The starting point for the paper is the abstraction method defined in [8], which presents a family of abstract domains that are useful when it is desired to connect storage elements (e.g., elements of arrays and lists) with numeric quantities. This paper reformulates that abstraction in a more general way—as a general method of abstracting a set of functions—which allows the basic idea from [8] to be applied more widely. Moreover, when the new formulation is compared with previously known ways of abstracting a set of functions, it yields more precise abstractions. We are just beginning to explore instantiations of the method that go beyond the ones used in [8].

We formalize a generic abstract-interpretation combinator, which abstract sets of functions of signature $D_1 \rightarrow D_2$ in a relational way, assuming the existence of abstractions A_1 and $A_2[n]$ for $\wp(D_1)$ and $\wp((D_2)^n)$, respectively (where A_1 is of finite cardinality n). The obtained abstract domain is precisely $A_2[n]$. In contrast to A_1, $A_2[n]$ may be a complex lattice (relational, infinite, and of infinite height), like the lattice of octagons [11] or convex polyhedra [5]. This *relational function-abstraction* is more precise than the classical approach described in the literature [4], because of its ability to represent relationships between the images of different elements mapped by a set of functions. For instance, consider a set of functions $F \subseteq U \rightarrow \mathbb{R}$ (that may represent a set of possible values for an array

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 186–202, 2005.
© Springer-Verlag Berlin Heidelberg 2005

of reals). If all $f \in F$ satisfy $f(u_1) = f(u_2)$, our abstraction is able to preserve this information in the abstract domain; in that precise sense it may be qualified as *relational* (which differs from the usual definition of [10]).

In terms of precision, the relational function-abstraction $A_2[n]$ lies in-between the classical function-abstraction $A_1 \to A_2$ and its disjunctive completion [3]. The important point is that $A_2[n]$ is still finitely representable in the same circumstances as $A_1 \to A_2$ (i.e., when $A_1 \to A_2$ is finitely representable, assuming a tabulated representation).

The contribution of the paper are as follows:

- we give an overview of the existing approaches to abstracting functions and relations and analyze the loss of information induced by them;
- we state our new approach to abstracting functions and compare it to existing ones in terms of expressiveness and implementability;
- we illustrate these different abstractions by considering their use in shape analysis; as a side-effect, we show how canonical abstraction can be partially recast in terms of a powerful combination of elementary abstractions, without resorting to the logical framework of [14].

In contrast to the domain construction and refinement approach (e.g., [12,6,7]), which operates on general lattices, our approach explicitly exploits the functional structure of concrete states.

The remainder of the paper is organized into four sections: Section 2 introduces some terminology and notation. Section 3 reviews the classical abstractions of functions of signature $D_1 \to D_2$ and relations between elements of D_1 and D_2 that were described in [4]. Section 4 describes relational function-abstraction. Section 5 presents related work and draws some conclusions.

2 Preliminaries

2.1 Lattices and Galois Connections

We denote by $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ a *lattice* defined by the set L and the partial order $\sqsubseteq$, where $\bot$, $\top$, $\sqcup$, and $\sqcap$ denote the smallest element, the greatest element, the least upper bound, and the greatest lower bound, respectively. Given any set D, the powerset $\wp(D)$ is a lattice ordered by set inclusion, and the set of functions $D \to L$ is a lattice ordered by the pointwise ordering: $f \sqsubseteq g \Leftrightarrow \forall d \in D : f(d) \sqsubseteq g(d)$. Given two lattices L_1 and L_2, $L_1 \times L_2$ is a lattice ordered componentwise, in which a pair (x_1, x_2) is identified with $\bot$ if either component is $\bot$. A function $f : L_1 \to L_2$ is *strict* and *total* if $f(\bot) = \bot \wedge f(x) = \bot \implies x = \bot$, *monotonic* if $x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$, and *additive* if $f(x \sqcup y) = f(x) \sqcup f(y)$. We denote these sets of functions by $L_1 \xrightarrow{\bot} L_2$, $L_1 \xrightarrow{\sqsubseteq} L_2$, and $L_1 \xrightarrow{\sqcup} L_2$, respectively. A lattice will be called a *flat lattice* if it is formed by a set of unordered elements to which a smallest element and a greatest element are added.

A Galois connection $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$ between two lattices C and A is defined by abstraction and concretization functions $\alpha : C \to A$ and $\gamma : A \to C$ that satisfy

$\forall x \in C, \forall y \in A : x \sqsubseteq_C \gamma(y) \iff \alpha(x) \sqsubseteq_A y$. In program analysis, C is most often the powerset of states $\wp(S)$. For any Galois connection: (*i*) $\gamma \circ \alpha$ is extensive (*i.e.*, greater than the identity function) and represents the information lost by the abstraction; (*ii*) α preserves $\sqcup$, and γ preserves $\sqcap$; (*iii*) α is one-to-one *iff* γ is onto *iff* $\gamma \circ \alpha$ is the identity; in this case, we use the notation $C \underset{\alpha}{\overset{\gamma}{\leftleftarrows\!\!\rightarrow}} A$. If $\gamma \circ \alpha$ is the identity, α loses no information and we will consider that C and A are isomorphic *from the information standpoint*, which is denoted by $C \simeq A$ (although γ may not be one-to-one).

Any Galois connection $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$ can be refined by considering the *disjunctive completion* of A [2], which corresponds to $\wp(A)$ equipped with an inclusion order that takes into account the order of A. This refinement allows the disjunction of abstract properties in A to be represented exactly, instead of using $\sqcup_A$, which usually loses information (*i.e.*, in general, one has $\gamma(x \sqcup_A y) \sqsupseteq \gamma(x) \sqcup_C \gamma(y)$). We denote by $\wp(A) \underset{\alpha_\vee}{\overset{\gamma_\vee}{\leftrightarrows}} A$ the Galois connection between the disjunctive completion of a lattice and itself. A is said to be *disjunctive* if it isomorphic to its disjunctive completion.

As mentioned in the introduction, Galois connections can be composed. We use $[\sigma, \varsigma]$ to denote the composition of a connection σ followed by a connection ς (so that we have $\alpha_{[\sigma,\varsigma]} = \alpha_\varsigma \circ \alpha_\sigma$).

The existence of a Galois connection between two lattices L_1 and L_2 defines the following pre-order: $L_1 \succeq L_2 \Leftrightarrow L_1 \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} L_2$. Given a concrete lattice C, the set of all (equivalence class of) lattices that abstract it, ordered by $\succeq$, is itself a lattice with top element C (see for instance Fig. 2).

2.2 Shape Analysis and Modeling Program States

This section provides background on the semantic domains used in shape analysis; this material will be used later in the paper to illustrate several aspects of the different approaches to abstracting sets of functions.

The aim of *shape analysis* is to analyze the properties of programs that manipulate heap-allocated storage and perform destructive updating of pointer-valued fields [14]. The goal is to recover shape descriptors that provide information about the characteristics of the data structures that a program's pointer variables can point to. Typically, work on shape analysis considers an imperative language that is equipped with an operational semantics defined using a transition relation between program states.

At a given control point, a program state $s \in S$ is defined by the values of the local variables and the heap. At each control point, the set of possible *concrete* properties on states is thus $\wp(S)$. The collecting semantics of programs is defined as a system of equations on the lattice of concrete properties.

We now describe two ways in which a state s can be modeled (*cf.* Fig. 1).

- The set-theoretic model is perhaps more intuitive. We consider a fixed set Cell of memory cells. The value of a pointer variable `z` is modeled by an element $z \in \mathrm{Cell} \cup \{\mathrm{nil}\}$, where nil denotes the null value. If cells have

Set-theoretic model			
Set of cells	Pointer variable `z`	Pointer field `n`	Real-valued field `x`
Cell	$z \in \text{Cell} \cup \{\text{nil}\}$	$n : \text{Cell} \rightarrow \text{Cell} \cup \{\text{nil}\}$	$x : \text{Cell} \rightarrow \mathbb{R}$
U	$z : U \rightarrow \mathbb{B}$	$n : U^2 \rightarrow \mathbb{B}$	$x : U \rightarrow \mathbb{R}$
Universe	Unary predicate	Binary predicate	Real-valued function
Logical model			

Fig. 1. Two models of a program state

a pointer-valued field `n`, the values of `n`-fields are modeled by a function $n : \text{Cell} \rightarrow \text{Cell} \cup \{\text{nil}\}$ that associates with each memory cell the value of the corresponding field.

– [14] models a state using the tools of logic: the set of cells is replaced by a universe U of individuals; the value of a program variable `z` is defined by a unary predicate on U; and the value of a field `n` is defined by a binary predicate on U^2. Integrity constraints are used to capture the fact that, for instance, a unary predicate z that represents what program variable `z` points to can have the value "true" for at most one memory cell [14].

A real-valued field `x` can be modeled by a real-valued function on U.

We use the term "predicate of arity n" for a Boolean function $U^n \rightarrow \mathbb{B}$. A predicate can also be seen as a relation belonging to $\wp(U^n)$.

We use $\mathcal{P}_n$ to denote the set of predicates symbols of arity n, and $\mathcal{R}$ to denote the set of real-valued function symbols. With such notation, the concrete state-space considered is:[1]

$$S = (U \rightarrow \mathbb{B})^{|\mathcal{P}_1|} \times (U^2 \rightarrow \mathbb{B})^{|\mathcal{P}_2|} \times (U \rightarrow \mathbb{R})^{|\mathcal{R}|} \tag{1}$$

A concrete property in $\wp(S)$ is thus a *relation between functions*.

Because U is of unbounded size, concrete properties belonging to $C = \wp(S)$ have to be abstracted. The idea behind canonical abstraction [14] is to partition U into a finite set of equivalence classes $U^\sharp$, and to introduce an *unknown* value 1/2 (or top element) to the Boolean set, yielding $\mathbb{T} = \{0, 1, 1/2\}$, so that a predicate $p : U \rightarrow \mathbb{B}$ is abstracted by an object $p^\sharp : U^\sharp \rightarrow \mathbb{T}$.

Example 1. In [14] and in Eqn. (1), the basic sets in use are the universe $D_1 = U$, the set of Booleans $D_2 = \mathbb{B}$, and the set of reals $D_3 = \mathbb{R}$. The universe U is partitioned using an equivalence relation $\simeq$, resulting in $U^\sharp = U/\simeq$. We use $\pi : U \rightarrow U^\sharp$ to denote the corresponding projection function. We then obtain a Galois connection

$$\wp(U) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (U^\sharp)^\top_\bot$$

[1] Eqn. (1) is really the concrete state-space that one would have if the techniques of [14] were combined with those of [8]. To simplify Eqn. (1), we have omitted nullary predicates, which would be used to model Boolean-valued variables, and nullary functions, which would be used to model real-valued variables.

where $(U^\sharp)_\perp^\top$ is the flat domain $U^\sharp$ completed with top and bottom elements, $\gamma(\perp) = \emptyset$, $\gamma(\top) = U$, and $\gamma(u^\sharp) = \pi^{-1}(u^\sharp)$. Booleans are not abstracted, *i.e.*, the domain $\wp(\mathbb{B})$ is abstracted by itself. In most cases, we will not consider the abstractions of reals. In the sequel, $U^\sharp$ will implicitely denote $(U^\sharp)_\perp^\top$. □

3 Classical Abstractions of Functions and Relations

We recall from [4] the classical abstractions of functions of signature $D_1 \rightarrow D_2$ and relations between elements of D_1 and D_2 that can be built from two Galois connections $\wp(D_1) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} A_1$ and $\wp(D_2) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} A_2$. Our notation is mainly taken from [4]. We also make some observations in Sec. 3.3 about exploiting the interplay between functions and relations to obtain suitable abstractions.

We first describe two useful isomorphisms used in the sequel.

– For any set D and lattice L, we have $(D \rightarrow L) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\wp(D) \xrightarrow{\perp,\sqsubseteq} L)$:

$$\alpha(F)(X) = \bigsqcup_{d \in X} F(d) \quad , \quad \gamma(F^\sharp)(d) = F^\sharp(\{d\})$$

This isomorphism allows to use directly the Galois connection $\wp(D_1) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} A_1$ to abstract the domain of functions $D_1 \rightarrow D_2$.

– For any sets D_1 and D_2, we have the isomorphism $\wp(D_1 \times D_2) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \wp_{\emptyset,\cup}(\wp(D_1) \times \wp(D_2))$, which allows to code relations on elements by relations on sets of elements:

$$\alpha(R) = \left\{ (X_1, X_2) \,\middle|\, \begin{array}{l} \forall d_1 \in X_1, \exists d_2 \in X_2 : (d_1, d_2) \in R \\ \forall d_2 \in X_2, \exists d_1 \in X_1 : (d_1, d_2) \in R \end{array} \right\}$$
$$\gamma(R^\sharp) = \{(x, y) \mid (\{x\}, \{y\}) \in R^\sharp\}$$

$\wp_{\emptyset,\cup}(L_1 \times L_2)$ denotes the set of relations $R \subseteq L_1 \times L_2$ that satisfies

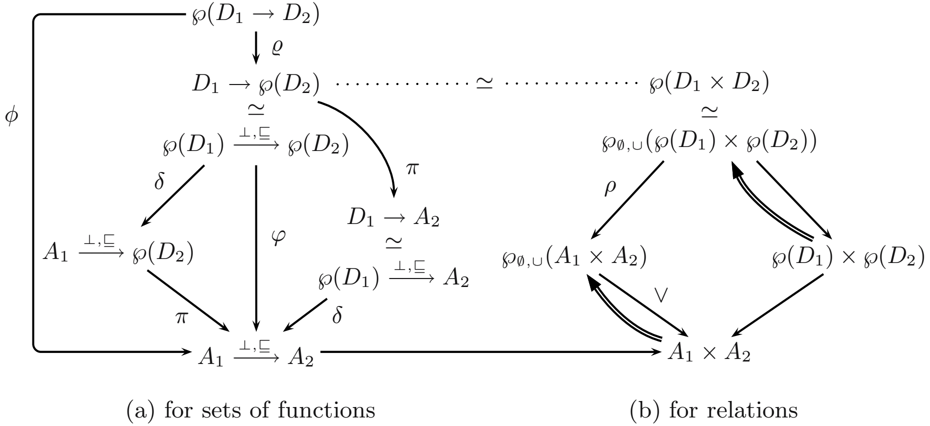
- $(\perp, x_2) \in R \implies x_2 = \perp$, as well as the converse.
- $(x_1, x_2) \in R \wedge (y_1, y_2) \in R \implies (x_1 \sqcup y_1, x_2 \sqcup y_2) \in R$ (this corresponds to a kind of upward-closure).

3.1 Classical Abstraction of a Functional Space

Fig. 2(a) shows classical abstract domains for sets of functions and their relationships. The first abstraction ϱ consists in abstracting a set of functions $F \subseteq D_1 \rightarrow D_2$ by a single function $F^\sharp : D_1 \rightarrow \wp(D_2)$ (or, equivalently, by a relation between D_1 and D_2):

$$\alpha_\varrho(F)(d_1) = \{f(d_1) \mid f \in F\} \quad , \quad f \in \gamma_\varrho(F^\sharp) \Leftrightarrow \forall d_1 : f(d_1) \in F^\sharp(d_1)$$

In words, $\alpha_\varrho(F)$ collects, for each argument in D_1, the set of its images by the functions in F. Consequently, this abstraction loses the possible relationship between $f(d_1)$ and $f(d'_1)$ that may hold for $f \in F$.



(a) for sets of functions (b) for relations

$L_1 \longrightarrow L_2$ means that $L_1 \succeq L_2$

$L_1 \Longleftarrow L_2$ means that L_1 is the disjunctive completion of L_2.

Fig. 2. Lattice of abstract domains for functions and relations

Example 2. A set of real-valued functions of signature $U \to \mathbb{R}$ is abstracted with α_ϱ by an element of $U \to \wp(\mathbb{R})$. Fig. 4 of Section 4 depicts concrete value A1, which is abstracted by value D $= \alpha_\varrho$(A1). The relationship $f(u_2) = f(u_1) + 1$ that holds in A1 is lost in D. □

One can then abstract the equivalent transformer $F : \wp(D_1) \xrightarrow{\perp,\sqsubseteq} \wp(D_2)$ with a function $F^\sharp : A_1 \xrightarrow{\perp,\sqsubseteq} A_2$ by abstracting both the domain and the codomain:

$$\alpha_\varphi(F) = \alpha_2 \circ F \circ \gamma_1 \quad , \quad \gamma_\varphi(F) = \gamma_2 \circ F^\sharp \circ \alpha_1$$

One can also abstract separately the domain (abstraction δ) and the codomain (pointwise abstraction π) to obtain two intermediate abstractions. The full composition of these Galois connections is the Galois connection $\phi = [\varrho, \varphi] = [\varrho, [\delta, \pi]] = [\varrho, [\pi, \delta]]$ between $\wp(D_1 \to D_2)$ and $A_1 \xrightarrow{\perp,\sqsubseteq} A_2$.

Let us take a closer look at the abstractions δ and π. Under δ, a function $F : D_1 \to \wp(D_2)$ is abstracted by a function $A_1 \to \wp(D_2)$ as follows: the image for an element $a \in A_1$ is computed by unioning together the images of the elements of D_1 represented by a:

$$\alpha_\delta(F)(a) = \bigcup_{\alpha_1(d)=a} F(d)$$

An application of this abstraction is illustrated in Fig. 4 ($E = \alpha_\delta(D)$). Under pointwise abstraction π, a function $F : D_1 \to \wp(D_2)$ is abstracted by representing the images of F by their abstraction in A_2: $\alpha_\pi(F) = \alpha_2 \circ F$.

Example 3. If we consider the basic Galois connections described in Example 1, a set of unary predicates of signature $U \to \mathbb{B}$ is abstracted with ϕ by an element of

$(U^\sharp)_\perp^\top \xrightarrow{\perp,\sqsubseteq} \wp(\mathbb{B})$. The fact that canonical abstraction [14] can represent exactly both false and true values of predicates (*i.e.*, it is conservative in both values) can be understood if you see unary predicates as ordinary functions abstracted this way. The same holds for binary predicates in $U^2 \to \mathbb{B}$. □

Remark 1. In practice, the abstraction A_1 of the domain D_1 of functions is often a flat lattice induced by a partitioning of D_1.

Remark 2. If $A_1 \xrightarrow{\perp,\sqsubseteq} A_2$ is to be implemented, each of its elements has to be finitely representable. This implies that A_2 should be finitely representable, and A_1 should be finite, as in Example 3; in this case $A_1 \xrightarrow{\perp,\sqsubseteq} A_2 \simeq (A_2)^n$, where n is the size of the partition used to define A_1.

It is not necessary to have an $n+1^{\text{st}}$ dimension to represent the image of $\perp_{A_1}$, which is $\perp_{A_2}$. When A_1 is built as in Example 3, the image of $\top_{A_1}$ does not carry any additional information, because $\gamma_1(\bigsqcup_{a_1 \in A_1 \setminus \{\top_{A_1}\}}) = \bigcup_{a_1 \in A_1 \setminus \{\top_{A_1}\}} \gamma_1(a_1) = D_1$. However, when the latter property does not hold, one should introduce one additional dimension to the abstract domain for the image of $\top_{A_1}$.

3.2 Classical Abstraction of a Relation

A binary relation between elements belonging to D_1 and D_2, respectively, is an element of $\wp(D_1 \times D_2)$. Fig. 2(b) shows classical abstractions for relations. Roughly speaking, the right-hand side of Fig. 2(b) abstracts a relation between concrete elements as a pair of sets, and then abstracts the pair of sets component-wise.

The left-hand side of Fig. 2(b) abstracts a relation between concrete elements using a relation between abstract elements. The abstraction $\wp_{\emptyset,\cup}(\wp(D_1) \times \wp(D_2)) \underset{\alpha_\rho}{\overset{\gamma_\rho}{\leftrightarrows}} \wp_{\emptyset,\cup}(A_1 \times A_2)$ is defined by:

$$\alpha_\rho(R) = \{(\alpha_1(X_1), \alpha_2(X_2)) \mid (X_1, X_2) \in R\} \tag{2}$$

$$\gamma_\rho(R^\sharp) = \{(X_1, X_2) \mid \exists (a_1, a_2) \in R^\sharp : X_1 \subseteq \gamma_1(a_1) \wedge X_2 \subseteq \gamma_2(a_2)\} \tag{3}$$

$\wp_{\emptyset,\cup}(A_1 \times A_2)$ can also be obtained by disjunctive completion of $A_1 \times A_2$. An observation similar to Remark 1 holds when choosing A_1 and A_2 for building $\wp_{\emptyset,\cup}(A_1 \times A_2)$.

Those principles seem natural when D_1 and D_2 are simple sets without structure, but the notation may seem rather heavy. However, the power of these combinators for relations is that they can be used when, for instance, D_1 and D_2 are sets of functions that are in turn abstracted using the principles described in Section 3.1.

Example 4. Considering the state-space S described in Equation (1), if we abstract functions with ϕ and relations over functions with ρ, we obtain the Galois connection

$$\wp(S) \leftrightarrows \wp_{\emptyset,\sqcup}\left(\left(U^\sharp \to \wp(\mathbb{B})\right)^{|\mathcal{P}_1|} \times \left((U^\sharp)^2 \to \wp(\mathbb{B})\right)^{|\mathcal{P}_2|} \times \left((U^\sharp \to \wp(\mathbb{R})\right)^{|\mathcal{R}|}\right)$$

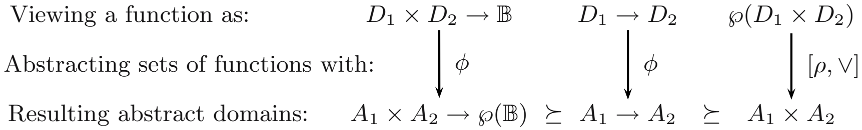


Fig. 3. Different ways of coding a set of functions, and the resulting abstractions

(Codomains of functions are not abstracted here, and abstract values are not finitely representable.) □

3.3 Exploiting Classical Abstractions

There is an interplay between the abstraction methods for functions and the abstraction methods for relations, because functions can be coded as relations and conversely. For instance, a function $D_1 \to D_2$ can be coded as a relation in $\wp(D_1 \times D_2)$. A relation in $\wp(D_1 \times D_2)$ can in turn be viewed as a Boolean function $D_1 \times D_2 \to \mathbb{B}$. Each view induces a different abstract domain using the abstractions of the previous sections, as shown in Fig. 3.

Example 5. Coming back to Example 3, if we view a data-structure field as a function $U \to U$ rather than a binary relation $U \times U \to \mathbb{B}$, as in [13,1], we will abstract $\wp(U \to U)$ by $U^\sharp \to \wp(U^\sharp) \simeq U^\sharp \times U^\sharp \to \mathbb{B}$.[2]

This abstraction is not conservative with respect to the value "true". That is, with this abstraction, "false" means "false", but "true" means "maybe true". An equivalent abstraction can be obtained when starting from $\wp(U^2 \to \mathbb{B})$ by abstracting $\wp(\mathbb{B})$ by conflating the values true and $\top$ [14, Section 8.2]. □

Generally speaking, abstraction methods for functions are more precise than abstraction methods for relations, as illustrated by Fig. 3. If we want to abstract relations in $\wp(D_1 \times D_2)$, viewing them as Boolean functions $D_1 \times D_2 \to \mathbb{B}$ and abstracting them with $A_1 \times A_2 \to \wp(\mathbb{B})$ instead of $\wp(A_1 \times A_2)$ allows to specify that some pairs of elements are definitely related.

Of course, the most suitable coding does not depend only on the induced abstraction, but also on the operations on functions (or relations) that are used for specifying the fixpoint equations to be solved. For instance, both concrete and abstract function application operations are defined in the most straightforward way when viewing a function as an element in $D_1 \to D_2$.

4 Relational Function-Abstraction

If we consider a set of functions of type $f : U \to \mathbb{R}$, and if we abstract it with the technique of Section 3.1 using convex polyhedra [5] for abstracting $\wp(\mathbb{R})$, we

[2] Here the domain U is abstracted by $U^\sharp$, and the codomain U by the more precise disjunctive completion $\wp(U^\sharp)$.

would obtain $U^\sharp \to \mathsf{Pol}[1] \simeq U^\sharp \to \mathsf{Interval}$. That is, we would associate to each abstract individual an interval. This is not what is proposed in [8], where an abstract value is a convex polyhedron, where each dimension corresponds to an abstract individual.

In this section, we formalize such an approach in more general terms; we analyze carefully the different kinds of abstractions that are performed; and we compare the approach to the classical approach described in Section 3.1.

4.1 Real-Valued Functions

To provide some intuition, we first instantiate the abstraction scheme for functions $f : U \to \mathbb{R}$. The universe U is partitioned by a projection function $\pi : U \to U^\sharp$. The cardinality of $U^\sharp$ is denoted by $|U^\sharp| = n$.

Our aim is the following: starting from the lattice $\wp(U \to \mathbb{R})$, we want to abstract it by a lattice of the form $\wp(U^\sharp \to \mathbb{R})$, for which many relational abstractions exist, instead of considering the lattice $U^\sharp \to \wp(\mathbb{R})$ obtained by the classical technique, which is not relational at all.

The abstraction proposed in [8] can be decomposed as follows:

$$\begin{array}{ccccccc} \wp(U \to \mathbb{R}) & \xrightarrow{\mu} & \wp(U^\sharp \to \wp(\mathbb{R})) & \xrightarrow{\eta} & \wp(U^\sharp \to \mathbb{R}) & \to & \mathsf{Pol}[n] \\ & & \simeq & & \simeq & & \\ & & \wp\big(\wp(\mathbb{R})^n\big) & & \wp(\mathbb{R}^n) & & \end{array}$$

The two isomorphisms mentioned in the above equation will be used later to encode functions as vectors: a function $f \in (U^\sharp \to D) \simeq D^n$ can be seen as a vector of elements of D by rewriting it as $\langle f(u_1^\sharp), \dots, f(u_n^\sharp)\rangle$.

Fig. 4 illustrates the three abstraction steps, which are defined in detail below.

1. $\wp(U \to \mathbb{R}) \underset{\alpha_\mu}{\overset{\gamma_\mu}{\leftrightarrows}} \wp(U^\sharp \to \wp(\mathbb{R}))$ is defined by

$$\alpha_\mu(\{f\}) = \{\langle X_1, \dots, X_n\rangle \mid X_i = f \circ \pi^{-1}(u_i^\sharp)\}\ ,\ \alpha_\mu(F) = \bigcup_{f \in F} \alpha_\mu(\{f\})$$
$$\gamma_\mu(\{f^\sharp\}) = \{f \mid \forall u : f(u) \in f^\sharp(\pi(u))\}\ ,\ \gamma_\mu(F^\sharp) = \bigcup_{f^\sharp \in F^\sharp} \gamma_\mu(\{f^\sharp\})$$

where f is lifted to sets in the expression $f \circ \pi^{-1}$. This Galois connection can be seen as the composition $\wp(U \to \mathbb{R}) \xrightarrow{[\varrho,\delta]} U^\sharp \to \wp(\mathbb{R})$ depicted in Fig. 2(a) refined by a disjunctive completion. This explains the fact that γ_μ preserves $\sqcup$.

Intuitively, this abstraction does not merge functions together; as illustrated by abstract value B1 in Fig. 4, it only merges the values $f(u)$ and $f(u')$ when u and u' are projected to the same abstract individual $u^\sharp$. Because $U^\sharp \to \wp(\mathbb{R})$ is ordered, a value in $\wp(U^\sharp \to \wp(\mathbb{R}))$ is completely characterized by its maximal elements with respect to the $U^\sharp \to \wp(\mathbb{R})$ ordering. (In Fig. 4, we only show maximal elements.)

2. $\underset{\wp(\wp(\mathbb{R})^n)}{\underset{\simeq}{\wp(U^\sharp \to \wp(\mathbb{R}))}} \xrightleftharpoons[\alpha_\eta]{\gamma_\eta} \underset{\wp(\mathbb{R}^n)}{\underset{\simeq}{\wp(U^\sharp \to \mathbb{R})}}$ is somewhat subtle:

$$\alpha_\eta(F) = \bigcup_{\langle X_1,\ldots,X_n\rangle \in F} X_1 \times \ldots \times X_n \tag{4}$$

$$\gamma_\eta(F^\sharp) = \{\langle X_1, \ldots, X_n\rangle \mid X_1 \times \ldots \times X_n \subseteq F^\sharp\} \tag{5}$$

   Note that in the right-hand-side lattice, $\wp(U^\sharp \to \mathbb{R})$, each dimension corresponds to a real instead of a set of reals. The subtle point in Eqn. (5) is that the set of vectors $F^\sharp$ is undercovered by a union of Cartesian products.
3. $\wp(\mathbb{R}^n) \overset{\gamma_{\mathsf{Pol}}}{\leftarrow} \mathsf{Pol}[n]$ is the abstraction of sets of vectors by convex polyhedra (in this particular case, because $\mathsf{Pol}[n]$ is not a complete lattice, the abstraction must be formalized using a weaker relationship than Galois connection). Other numerical lattices could be used in place of $\mathsf{Pol}[n]$.

To focus on the abstraction of the domain U, in the following discussion and in Fig. 4 we ignore the abstraction of codomain $\wp(\mathbb{R})$—*i.e.*, we assume that the codomain is not abstracted. (Using the notation of Fig. 2(a), we are redefining ϕ to be the abstraction $[\varrho, \delta]$.)

Intuitively, the composition of abstractions 1. and 2. allows capturing relationships between the images of the different arguments of the functions, when they belong to different equivalence classes. In contrast, with the abstraction ϕ of Fig. 2(a), such relationships are lost (due to the abstraction ϱ).

Example 6. In Fig. 4, the abstraction of concrete value A1 using relational function-abstraction is abstract value C. C concretizes to concrete value A3. Inspection of A3 reveals that abstract value C preserves from A1 the properties $f(u_2) = f(u_1) + 1$ and $\forall u \in \{u_3, u_4\} : f(u_1) + 3 \leq f(u) \leq f(u_1) + 4$. However, C loses the property of A1 that whenever $f(u_1) = 1$, $f(u_3) = f(u_4)$.

In contrast, by using the abstraction ϕ of Section 3.1, we obtain the abstract (functional) value E, from which one can only deduce weaker properties, as shown by its concretization A4. A3 is of cardinality 8, whereas A4 is of cardinality 36. □

To study the loss of information induced by abstraction η (*i.e.*, abstraction-step 2. above), let us consider the expression $\gamma_\eta \circ \alpha_\eta$:

$$\gamma_\eta \circ \alpha_\eta(F) = \left\{ \langle X_1, \ldots, X_n\rangle \;\middle|\; X_1 \times \ldots \times X_n \subseteq \bigcup_{\langle Y_1,\ldots,Y_n\rangle \in F} Y_1 \times \ldots \times Y_n \right\}$$

In words, this means that $(\gamma_\eta \circ \alpha_\eta)(F)$ adds to F Cartesian products that underapproximate unions of Cartesian products.

Example 7. In Fig. 4, we have $\mathrm{B2} = (\gamma_\eta \circ \alpha_\eta)$ (B1). The information that whenever $f(u_1) = 1$, $f(u_3) = f(u_4)$ is lost. More generally, η will lose relational information about $f(u)$ and $f(u')$ when u and u' are merged together. □

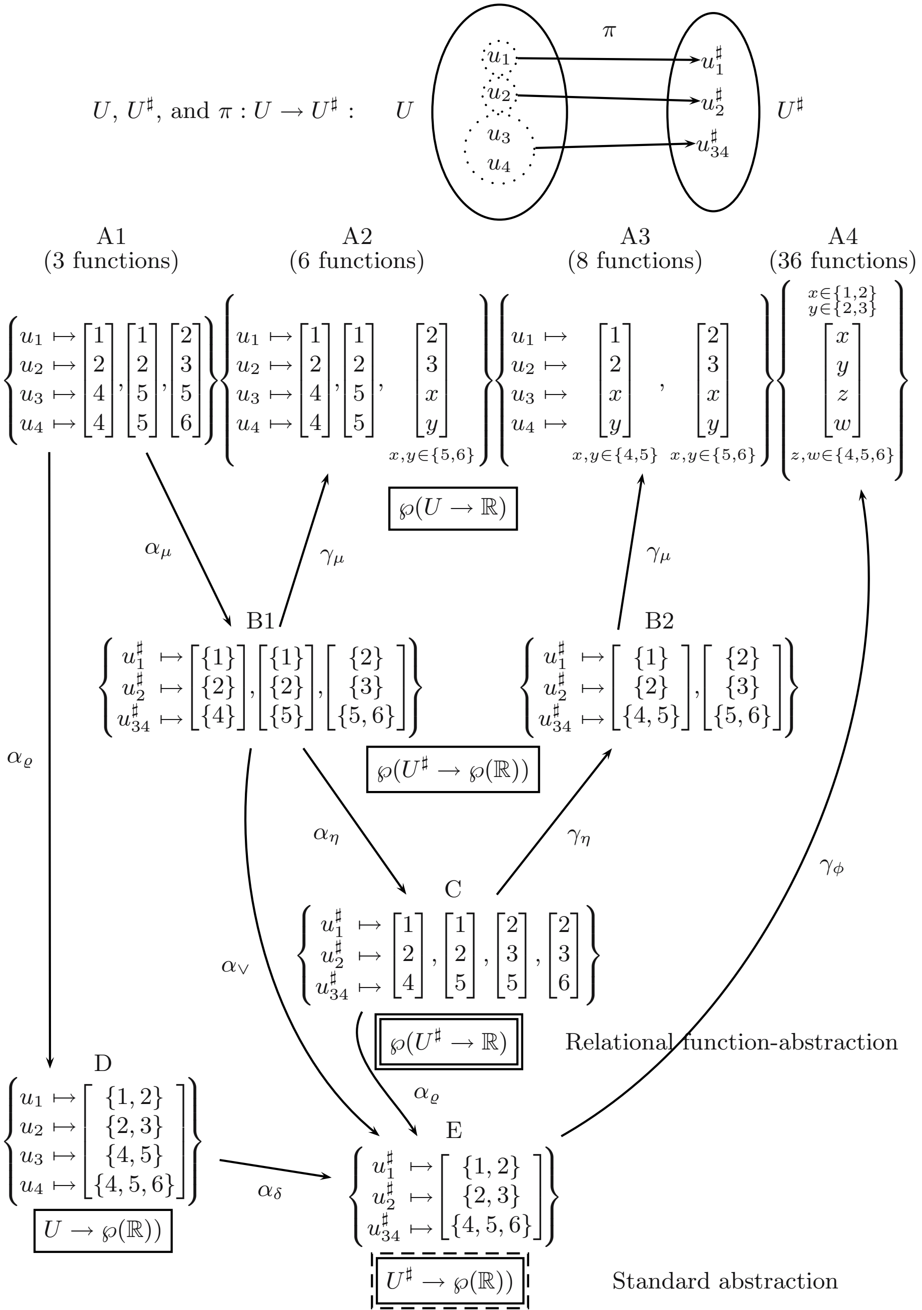


Fig. 4. Different abstractions of the concrete set of functions A1 and the loss of information induced by them (shown by concrete values A2, A3, and A4). Abstract value C (whose concretization is A3) is the abstract value obtained with relational function-abstraction Φ, whereas abstract value E (whose concretization is A4) is the abstract value obtained by the abstraction ϕ of Section 3.1.

4.2 Generalization

In this section, we generalize the results of Section 4.1 by considering the abstraction of functions of signature $D_1 \to D_2$. We also analyze the effect of the abstracting codomain D_2.

Our assumptions are as follows: Suppose that we have a Galois connection $\wp(D_1) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} A_1$, where A_1 is a finite lattice with $|A_1| = n$. Actually, this assumption is only needed for the third abstraction step of Def. 1. In addition, suppose that for any k, we have a Galois connection $\wp((D_2)^k) \leftrightarrows A_2[k]$, where $A_2[k]$ is a k-dimensional abstract domain. We use A_2 to denote $A_2[1]$. We also assume that $\forall k \geq 1 : A_2[k+1] \succeq A_2[k] \times A_2[1]$. If the inequality is actually an equality, $A_2[k] = (A_2)^k$ is said to be *not relational* [10] (for instance, intervals on reals). If the inequality is strict, $A_2[k]$ is *relational* (for instance, convex polyhedra).

Definition 1 (Relational function-abstraction Φ). *The* relational function-abstraction Φ *is defined by the composition of the following three abstractions:*

$$\wp(D_1 \to D_2) \xrightarrow{\mu} \wp(A_1 \xrightarrow{\bot,\sqsubseteq} \wp(D_2)) \xrightarrow{\eta} \wp(A_1 \backslash \{\bot\} \to D_2) \xrightarrow{\alpha_2[|A_1|]} A_2[|A_1|]$$

where

1. $\wp(D_1 \to D_2) \underset{\alpha_\mu}{\overset{\gamma_\mu}{\leftrightarrows}} \wp(A_1 \xrightarrow{\bot,\sqsubseteq} \wp(D_2))$ *is defined by*

$$\alpha_\mu(F) = \bigcup_{f \in F} \{f^\sharp \mid \forall a_1 : f^\sharp(a_1) = f(\gamma_1(a_1))\} \tag{6}$$

$$\gamma_\mu(F^\sharp) = \bigcup_{f^\sharp \in F^\sharp} \{f \mid \forall d_1 : f(d_1) \in f^\sharp(\alpha_1(d_1))\} \tag{7}$$

2. $\wp(A_1 \xrightarrow{\bot,\sqsubseteq} \wp(D_2)) \underset{\alpha_\eta}{\overset{\gamma_\eta}{\leftrightarrows}} \wp(A_1 \backslash \{\bot\} \to D_2)$ *is defined by*[3]

$$\alpha_\eta(F) = \bigcup_{f \in F} \{f^\sharp \mid \forall a_1 : f^\sharp(a_1) \in f(a_1)\} \tag{8}$$

$$\gamma_\eta(F^\sharp) = \{f \mid \forall f^\sharp \in (A_1 \to D_2) : \Big(\forall a_1, f^\sharp(a_1) \in f(a_1)\Big) \Rightarrow f^\sharp \in F^\sharp\} \tag{9}$$

3. The last abstraction is the abstraction of sets of D_2-valued vectors by $A_2[n]$.

Notice that if we drop the assumption that A_2 is finite, we still provide an original method for abstracting $\wp(D_1 \to D_2)$ with $\wp(A_1 \backslash \{\bot\} \to D_2)$. We identify below an important class of properties preserved by this abstraction, generalizing the properties mentioned in Example 6.

[3] We confess that Eqn. (9), which is derived from the definition of α_η, is difficult to understand. However, this formulation is more general than Eqn. (5); *i.e.* it can be used when A_1 is not finite.

Theorem 1 (Relational properties preserved by the abstraction $[\mu, \eta]$). *Let $P_1 \subseteq (D_1)^2$ and $P_2 \subseteq (D_2)^2$ be binary relations on D_1 and D_2, and let $\Psi_{[P_1,P_2]} \in \wp(D_1 \to D_2)$ be a property on functions defined by*

$$\Psi_{[P_1,P_2]} = \{f : D_1 \to D_2 \mid \forall (d_1, d_1') \in P_1, (f(d_1), f(d_1')) \in P_2\}.$$

Assume that P_1 is preserved by the abstraction $\wp(D_1) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} A_1$ as follows:

$$(d_1, d_1') \in P_1 \Rightarrow (\gamma_1 \circ \alpha_1)(d_1) \times (\gamma_1 \circ \alpha_1)(d_1') \subseteq P_1.$$

Let $F \subseteq \Psi_{[P_1,P_2]}$. Then $(\gamma_{[\mu,\eta]} \circ \alpha_{[\mu,\eta]})(F) \subseteq \Psi_{[P_1,P_2]}$; i.e., property $\Psi_{[P_1,P_2]}$ is preserved by the abstraction $[\mu, \eta]$.

Example 8. Coming back to Example 6 and Fig. 4, let $P_1 = \{(u_1, u_3), (u_1, u_4)\}$ and $P_2 = \{(x, y) \mid x + 3 \leq y \leq x + 4\}$. Then the property $\Psi_{[P_1,P_2]}$ is $\forall u \in \{u_3, u_4\} : f(u_1) + 3 \leq f(u) \leq f(u_1) + 4$, which is satisfied by the set of functions A1. Because P_1 is preserved by the abstraction $\wp(U) \leftrightarrows U^\sharp$, and A3 is equal to $(\gamma_{[\mu,\eta]} \circ \alpha_{[\mu,\eta]})(A1)$, A3 also satisfies $\Psi_{[P_1,P_2]}$.

The above theorem is generalizable to k-ary relations P_1 and P_2 for $k \leq |A_1|$. Also, the theorem implies that if, in addition, the relation P_2 is preserved by the abstraction $\alpha_2[|A_1|]$, then the property $\Psi_{[P_1,P_2]}$ is preserved by the full relational function-abstraction Φ. This is the case in the example above if one uses octagons, for instance.

Proof. Let $P_1^\sharp = \{(a_1, a_1') \in (A_1)^2 \mid \exists d_1 \in \gamma_1(a_1), \exists d_1' \in \gamma_1(a_1') : (d_1, d_1') \in P_1\}$. Notice that, due to the assumption on P_1, P_1 is fully defined by $P_1^\sharp$. Then

$$\alpha_\mu(\Psi_{[P_1,P_2]}) = \left\{ f : A_1 \to \wp(D_2) \;\middle|\; \begin{array}{c} \forall (a_1, a_1') \in P_1^\sharp, \forall (d_2, d_2') \in f(a_1) \times f(a_1') : \\ (d_2, d_2') \in P_2 \end{array} \right\}$$

$$\alpha_{[\mu,\eta]}(\Psi_{[P_1,P_2]}) = \{f : A_1 \to D_2 \mid \forall (a_1, a_1') \in P_1^\sharp : (f(a_1), f(a_1')) \in P_2\}$$

$$\gamma_\eta \circ \alpha_{[\mu,\eta]}(\Psi_{[P_1,P_2]}) = \left\{ f : A_1 \to \wp(D_2) \;\middle|\; \begin{array}{c} \forall f^\sharp : A_1 \to D_2 : \\ \Big(\forall a_1, f^\sharp(a_1) \in f(a_1)\Big) \Rightarrow \\ \Big(\forall (a_1, a_1') \in P_1^\sharp : (f^\sharp(a_1), f^\sharp(a_1')) \in P_2\Big) \end{array} \right\}$$

We now show that $\gamma_\eta \circ \alpha_{[\mu,\eta]}(\Psi_{[P_1,P_2]}) \subseteq \alpha_\mu(\Psi_{[P_1,P_2]})$. Let $f \in \gamma_\eta \circ \alpha_{[\mu,\eta]}(\Psi_{[P_1,P_2]})$, and $(a_1, a_1') \in P_1^\sharp$. We have that for any $f^\sharp : A_1 \to D_2$ such that $\forall a, f^\sharp(a) \in f(a)$, then $(f^\sharp(a_1), f^\sharp(a_1')) \in P_2$. It follows that $\forall (d_2, d_2') \in f(a_1) \times f(a_1')) : (d_2, d_2') \in P_2$, which proves that $f \in \alpha_\mu(\Psi_{[P_1,P_2]})$. It is easy to show that $\gamma_\mu \circ \alpha_\mu(\Psi_{[P_1,P_2]}) \subseteq \Psi_{[P_1,P_2]}$. Now, because $F \subseteq \Psi_{[P_1,P_2]}$ and $(\gamma_{[\mu,\eta]} \circ \alpha_{[\mu,\eta]})$ is monotone, the desired relationship holds:

$$(\gamma_{[\mu,\eta]} \circ \alpha_{[\mu,\eta]})(F) \subseteq (\gamma_{[\mu,\eta]} \circ \alpha_{[\mu,\eta]})(\Psi_{[P_1,P_2]}) \subseteq \Psi_{[P_1,P_2]}. \qquad \square$$

Example 9. It is instructive to illustrate relational function-abstraction Φ for the case of Boolean functions in $U \to \mathbb{B}$, assuming that the codomain ($\mathbb{B}$) is not abstracted. We obtain $\wp(U^\sharp \to \mathbb{B}) \simeq \wp(\mathbb{B}^n)$; *i.e.*, an abstract value is a set of bit-vectors, or, equivalently, a propositional formula. If we use the abstraction

ϕ, we obtain instead $U^\sharp \rightarrow \wp(\mathbb{B}) \simeq \wp(\mathbb{B})^n$; *i.e.*, an abstract value is a trivector or a single monomial. This means that in this specific case the abstraction $\Phi = [\mu, \eta]$ reduces to the disjunctive completion μ of the abstraction $[\varrho, \delta] = \phi$. In particular, the abstraction η does not lose any information. This property is not true in general, as shown by Example 7. □

We now compare the relational function-abstraction Φ defined above with ϕ, the traditional approach to abstracting sets of functions, and its disjunctive completion.

Theorem 2. *We have the following relationships:*

$$\wp(A_1 \xrightarrow{\perp,\sqsubseteq} A_2) \;\succeq\; A_2[|A_1|] \;\succeq\; (A_1 \xrightarrow{\perp,\sqsubseteq} A_2)$$

The first inequality reduces to an equality iff $A_2[|A_1|]$ is disjunctive. The second inequality reduces to an equality iff $A_2[|A_1|]$ is not relational.

Proof. Let $n = |A_1|$ be the cardinality of A_1. We have the isomorphism $(A_1 \xrightarrow{\perp,\sqsubseteq} A_2) \simeq (A_2)^n$. By hypothesis, $\forall k \geq 1 : A_2[k] \succeq (A_2)^k$.

As a consequence, $A_2[n] \succeq (A_2)^n$, which corresponds to the second inequality of the theorem. The equality and strict-inequality cases follow from the definition of a relational lattice.

We denote by $A_2[n] \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A_2)^n$ the Galois connection corresponding to the inequality $A_2[n] \succeq (A_2)^n$. We now define the Galois connection $\wp((A_2)^n) \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} A_2[n]$ with

$$\alpha'(X) = \sqcup\{\gamma(a) \mid a \in X\} \quad \text{and} \quad \gamma'(Y) = \{a \in (A_2)^n \mid \gamma(a) \sqsubseteq Y\}$$

One can easily check that this defines a Galois connection. This proves the first inequality of the theorem. The equality and strict-inequality cases follow from the definition of a disjunctive lattice (*cf.* Section 2.1) and the fact that $\wp(A_1 \xrightarrow{\perp,\sqsubseteq} A_2)$ is trivially disjunctive. □

Implementability Issues. An abstract lattice, even if not implementable, is interesting in so far as it may be used as a semantic domain in an abstraction chain. Here, adopting a pragmatic standpoint, we are interested in knowing when the three abstract domains $\wp(A_1 \xrightarrow{\perp,\sqsubseteq} A_2)$, $A_2[|A_1|]$, and $(A_1 \xrightarrow{\perp,\sqsubseteq} A_2)$ can be used in practice, *i.e.*, when their elements are finitely representable. For the sake of discussion, assume that $A_1 \xrightarrow{\perp,\sqsubseteq} A_2$ is finitely representable by an argument/value table (*e.g.*, A_1 is finite and A_2 is finitely representable). The domain $\wp(A_1 \xrightarrow{\perp,\sqsubseteq} A_2)$ is finitely representable only if A_2 is a finite lattice, or an infinite lattice that does not contain infinite subsets of incomparable elements (or anti-chains). In contrast, the relational function-abstraction $A_2[|A_1|]$ is always finitely representable under our assumptions. In particular,

- $A_2[|A_1|]$ is finitely representable when A_2 is a finite-height lattice. In this case it is also finite-height.
- $A_2[|A_1|]$ is finitely representable even when A_2 is not a finite-height lattice, and can be used in practice, provided that A_2 is equipped with a widening operator.

For instance, if $D_2 = \mathbb{R}$, $A_2[n]$ can be the relational lattice of octagons [11] or convex polyhedra [5]. If $D_2 = \Sigma^*$ is a language over an alphabet Σ, $A_2[n]$ can be the relational lattice $\mathsf{Reg}(\Sigma^n)$ of regular languages over vectors of letters equipped with a suitable widening operator. Neither of these lattices are finite-height.

The disussion above assumed that only argument/value tabular representations are available for functions in $A_1 \to A_2$. Of course, in particular cases, more efficient representations may be available that exploit the underlying structure of the domain and/or codomain. For instance, regular transducers are a very effective representation of (a subset of) functions on $\Sigma^* \to \Sigma^*$, which has both infinite domain and codomain.

4.3 Abstracting Relations over Functions with Relational Function-Abstraction

The strength of relational lattices is their ability to abstract a powerset of Cartesian products more precisely than the abstractions that were discussed in Section 3.2, as illustrated by the following example:

Example 10. Suppose that we want to abstract relations between vectors (*i.e.*, the concrete domain is $C = \wp(\mathbb{R}^n \times \mathbb{R}^m))$, where sets of vectors $\wp(\mathbb{R}^n)$ and $\wp(\mathbb{R}^m)$ are abstracted by convex polyhedra. The abstraction ρ of Fig. 2(b) results in the abstract domain $\wp(\mathsf{Pol}[n] \times \mathsf{Pol}[m])$, which is not finitely representable. The abstraction $[\rho, \vee]$ results in $\mathsf{Pol}[n] \times \mathsf{Pol}[m]$, which does not capture relationships between pairs very precisely. It is well-known that the most precise way to abstract C is to use the relational lattice $\mathsf{Pol}[n+m]$. □

A similar phenomenon arises when abstracting relations over functions with relational function-abstraction. If we want to abstract a relation in $\wp\big((D_1 \to D_2)\times(D_1 \to D_2)\big)$ (*i.e.*, a relation between functions that share the same domain and codomain), we should use $A_2[2 \cdot |A_1|]$. In other words, we can exploit the set-isomorphism $(D_1 \to D_2)^2 = D_1 \to (D_2)^2$, and then apply relational function-abstraction:

$$\wp\big((D_1 \to D_2) \times (D_1 \to D_2)\big) = \wp\big(D_1 \to (D_2)^2\big) \leftrightarrows A_2[2 \cdot |A_1|]$$

Example 11. Coming back to Eqn. (1), let us illustrate the principles described in this paper to obtain a finitely representable abstract domain for $\wp(S)$:

$$\wp(S) = \wp\left((U \to \mathbb{B})^{|\mathcal{P}_1|} \times (U^2 \to \mathbb{B})^{|\mathcal{P}_2|} \times (U \to \mathbb{R})^{|\mathcal{R}|}\right) \quad (10)$$

$$= \left((U \to \mathbb{B})^{|\mathcal{P}_1|} \times (U^2 \to \mathbb{B})^{|\mathcal{P}_2|}\right) \to \wp\left(U \to \mathbb{R}^{|\mathcal{R}|}\right) \quad (11)$$

$$\leftrightarrows \left(\left(U^\sharp \to \wp(\mathbb{B})\right)^{|\mathcal{P}_1|} \times \left((U^\sharp)^2 \to \wp(\mathbb{B})\right)^{|\mathcal{P}_2|}\right) \xrightarrow{\perp,\sqsubseteq} \mathsf{Pol}[\,|\mathcal{R}| \cdot |U^\sharp|\,] \quad (12)$$

The function in Eqn. (11) is abstracted using the function-abstraction φ of Fig. 2, the abstraction $[\rho, \vee]$ being used for the domain and the relational function-abstraction Φ being used for the codomain. Intuitively, we associate a convex polyhedra to each vector of abstract Boolean functions. □

5 Related Work and Conclusions

We formalized in this paper a generic abstract-interpretation combinator, which abstracts sets of functions $D_1 \rightarrow D_2$ in a relational way, assuming the existence of abstractions A_1 and $A_2[n]$ for $\wp(D_1)$ and $\wp((D_2)^n)$, respectively. Viewed from another angle, we have shown how to give a new semantics—in terms of *sets of functions*—to some previously known abstract lattices, such as octagons and convex polyhedra. This was achieved by developing nonstandard concretization functions for such domains. As an intermediate step, we formalized the abstraction of sets of functions $D_1 \rightarrow D_2$ by sets of functions $A_1 \rightarrow D_2$ and identified a class of properties preserved by this abstraction.

In terms of precision, the abstract domains that we obtain from relational function-abstraction lie in-between the classical function-abstraction $A_1 \rightarrow A_2$ and its disjunctive completion. The important point is that the abstract domains obtained from relational function-abstraction are finitely representable in more general circumstances than the disjunctive completions of classical function-abstractions. In fact, they are finitely representable in the same circumstances as classical function-abstraction (i.e., when $A_1 \rightarrow A_2$ is finitely representable with a tabulated representation).

We focused in this paper on the compositional construction of abstract domains and ignored algorithmic issues, as well as the choice of basic abstract domains. The relational function-abstraction described in the paper has actually been implemented in [8] in the shape-analysis framework of [14], for abstracting real-valued fields of dynamically allocated data structures. More recently, [9] has addressed the problem of abstracting arrays of reals, viewed as functions of signature $[0..n] \rightarrow \mathbb{R}$. Both [8] and [9] address the difficult problem of abstracting the domain U of a function space in a suitable way. It appears than using a fixed partition of U is useless; instead, [8] and [9] support dynamic partitioning of U.

We compared our solution to the classical solutions described in [4] and their refinement with the disjunctive-completion method. We review here other refinement methods. The tensor product of [12] combines in a relational way two different abstract domains L_1 and L_2 that abstract the same concrete domain C; it is denoted $L_1 \otimes L_2$. The tensor product satifies the equation $\wp(S_1) \otimes \wp(S_2) = \wp(S_1 \times S_2)$ for powersets, and extends such an operation to more general lattices. The reduced cardinal power [2] and reduced relative power [6] combine L_1 and L_2 in a different way, by considering lattices of functions $L_1 \rightarrow L_2$, which captures *dependencies*, or in a more logical setting, *implications* [7]. In the particular case where $L_1 = L_2 = L$, this refinement allows to capture *autodependencies*, which is in general incomparable with the disjunctive completion of L. We did not fully explore whether the above refinements can

be used to generate relational function-abstraction Φ from classical abstractions for functions. However, we believe that even if it were possible, the construction would be more complicated than our approach:

– the functional structure of concrete states would not be exploited;
– the "syntactic" structure of the obtained abstract lattice ($L_1 \otimes L_2$ or $L_1 \rightarrow L_2$) would be quite different from $A_2[n]$, even if an isomorphism exists.

References

1. D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. ACM Press, 1990.
2. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM Symposium on Principles of Programming Languages, POPL'79*, San Antonio, January 1979.
3. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.
4. P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proc. of the 1994 Int. Conf. on Computer Languages*, Toulouse, France, May 1994. IEEE Computer Society Press.
5. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages, POPL'78*, Tucson (Arizona), January 1978.
6. R. Giacobazzi and F. Ranzato. The reduced relative power operation on abstract domains. *Theoretical Computer Science*, 216, 1999.
7. R. Giacobazzi and F. Scozzari. A logical model for relational abstract domains. *ACM Trans. Program. Lang. Syst.*, 20(5), 1998.
8. D. Gopan, F. DiMaio, N.Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Int. Conf. on Tools and Algs. for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, 2004.
9. D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *32th ACM Symposium on Principles of Programming Languages, (POPL'05)*. ACM press, January 2005.
10. N. Jones and S. Muchnick. Complexity of flow analysis, inducive assertion synthesis, and a language due to Dijkstra. In N. Jones and S. Muchnick, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
11. A. Miné. The octagon abstract domain. In *AST'01 in Working Conference on Reverse Engineering 2001*. IEEE CS Press, October 2001.
12. F. Nielson. Tensor products generalize the relational data flow analysis method. In *Fourth Hungarian Computer Science Conference*, 1985.
13. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Symposium on Principles of Programming Languages (POPL'96)*, pages 16–31, New York, NY, January 1996. ACM Press.
14. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3), 2002.

Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis*

Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi

Programming Research Laboratory,
School of Computer Science and Engineering,
Seoul National University
{dreameye, jaehwang, netj, kwang}@ropas.snu.ac.kr

Abstract. We present our experience of combining, in a realistic setting, a static analyzer with a statistical analysis. This combination is in order to reduce the inevitable false alarms from a domain-unaware static analyzer. Our analyzer named Airac(Array Index Range Analyzer for C) collects all the true buffer-overrun points in ANSI C programs. The soundness is maintained, and the analysis' cost-accuracy improvement is achieved by techniques that static analysis community has long accumulated. For still inevitable false alarms (e.g. Airac raised 970 buffer-overrun alarms in commercial C programs of 5.3 million lines and 737 among the 970 alarms were false), which are always apt for particular C programs, we use a statistical post analysis. The statistical analysis, given the analysis results (alarms), sifts out probable false alarms and prioritizes true alarms. It estimates the probability of each alarm being true. The probabilities are used in two ways: 1) only the alarms that have true-alarm probabilities higher than a threshold are reported to the user; 2) the alarms are sorted by the probability before reporting, so that the user can check highly probable errors first. In our experiments with Linux kernel sources, if we set the risk of missing true error is about 3 times greater than false alarming, 74.83% of false alarms could be filtered; only 15.17% of false alarms were mixed up until the user observes 50% of the true alarms.

1 Introduction

When one company's software quality assurance department started working with us to build a static analyzer that automatically detect buffer overruns[1] in

* This work was supported by Brain Korea 21 Project of Korea Ministry of Education and Human Resources, by IT Leading R&D Support Project of Korea Ministry of Information and Communication, by Korea Research Foundation grant KRF-2003-041-D00528, and by National Security Research Institute.

[1] Buffer overruns happen when an index value is out of the target buffer size. They are common bugs in C programs and are main sources of security vulnerability. From 1/2[2] to 2/3[1] of security holes are due to buffer overruns.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 203–217, 2005.
© Springer-Verlag Berlin Heidelberg 2005

their C softwares, they challenged us with three goals: they hoped the analyzer 1) to be sound, detecting all possible buffer overruns; 2) to have a reasonable cost-accuracy balance; and 3) not to assume a particular set of programming style about the input C programs because they handle a wide spectrum of C softwares to be embedded in various electronic devices. Building a realistic C buffer-overrun analyzer that satisfies all the three requirements was a hard challenge. In the literature, we have seen impressive static analyzers yet their application targets seem to allow them to drop one of the three requirements [7,4,13,9]. The major challenge is how to reduce the number of inevitable false alarms from a realistic, sound static analyzer that cannot assume a particular style for the input C programs.

In respond to the challenge, we decided to try the following path: design a sound static analysis whose accuracy is stretched to a point where the analysis cost remains acceptable, then use a statistical post analysis in order to sift out alarms that are probable to be false. The analyzer named Airac(Array Index Range Analyzer for C) collects all the true buffer-overrun points in ANSI C programs. The soundness is maintained, and the analysis' cost-accuracy balance is stroke with techniques that static analysis community has long accumulated. Now for still inevitable false alarms, which are always apt for particular C programs, we use a statistical post analysis. The statistical analysis, given the analysis results (alarms), sifts out some alarms that are probable to be false. It estimates the probability of each alarm being true. The probabilities are used in two ways: 1) only the alarms that have true-alarm probabilities higher than a threshold are reported to the user. The threshold is determined by the user-provided ratio of the risk of silencing true alarms to that of false alarming. 2) By sorting the alarms to be reported in descending order, it allows the user to examine highly probable alarms first.

Airac targets the full set of ANSI C constructs as indexing expressions: from simple arithmetics to arbitrary expressions involving function calls, pointer arithmetics, and aliases. Airac handles buffers that are dynamically allocated consecutive memory cells of dynamic lengths as well as static arrays. "*buffer overrun*" happens when an index value denotes an address outside the target buffer area. Striking a cost-accuracy balance of Airac is done by the following techniques: for accuracy improvement we use narrowing after widening, flow-sensitivity, polyvariance, context pruning (an instance of trace partitioning[11]) and static loop unrolling. For cost reduction, we used stack obviation (removal of the stack from our abstract state), selective memory join (point-wise join for abstract memory is applied only to the changed entries), and wait-at-join (a work-list iteration does not continue to pass a join point until all threads arrive). For commercial C programs of 5.3 million LOC, Airac raises 970 buffer-overrun alarms, among which 233 alarms are true. For some parts of the Linux kernel of 18,760 LOC, Airac raises 26 alarms, among which 16 are true.

The statistical method aiming to sift out false alarms is designed by the Bayesian data analysis framework[8], implemented by the Monte Carlo method [12], and parameterized by a simple decision theory[3]. We define a conditional

probability formula for an alarm to be true given the set of symptoms observed for the alarm. This probability formula has parameter probabilities, whose distributions are determined by Bayesian analysis from the "training set" (or "past experience knowledge"). The parameter probabilities are obtained by the Monte Carlo method. The training set, which consists of alarms and their conditional probabilities of having symptoms given that they are either true or false, is obtained by running our analyzer for a set of Linux kernel, and textbook C programs and manually classifying the alarms into either true or false. Having computed the probability of each alarm being true, we report only the alarms that have the true-alarm probabilities higher than a threshold. The threshold is determined by the user-provided ratio of the risk of silencing true alarms to that of raising false alarms. The ratio, for each alarm, determines the expected risks of silencing it or alarming it. The action with a smaller risk is chosen. This statistical engine's effectiveness is promising. If the user set the risk of missing true error is 3 times greater than false alarming, then 74.83% of false alarms could be sifted out. Meanwhile, by ranking the alarms by higher probabilities and examining from the top, the user encounters only 15.17% of false alarms until he or she reaches 50% of the true ones.

2 Airac, the Analyzer

Airac is an abstract interpreter. To find out all possible buffer overruns in programs, Airac considers all states which may occur during programs executions. It computes a sound approximation of dynamic program states occurring at each program point and reports possible buffer overruns by examining the approximate states.

A concrete array block is abstracted as a triple that consists of abstract base address, offset, and size. Abstract base address is one for each memory allocation site in C programs. Abstract offset and size are integer intervals. For example, for the following C code:

```
int p[5];
int *q = p + 3;
*(q+3) = 1;
```

The pointer `p`'s abstract value is $\langle l, [0,0], [5,5] \rangle$ where name l is the abstract base address for the declared array. $[0,0]$ and $[5,5]$ are respectively the current offset and size as intervals. After the pointer arithmetic, `q` is initialized as $\langle l, [3,3], [5,5] \rangle$; then the value of `q+3` is $\langle l, [6,6], [5,5] \rangle$ whose offset exceeds its size, where our analyzer raises a buffer overrun alarm.

2.1 Semantics and Its Abstraction

C program's collecting semantics is defined as the set of transition sequences of machine states. A machine state is a tuple of a program point, data stack, environment, memory, and control stack (dump). A C program's semantics is the

least fixed point of the following function that transforms the machine transition traces:

$$\mathcal{F} : 2^{Machine^{\omega}} \rightarrow 2^{Machine^{\omega}}$$
$$\mathcal{F}(X) = \{m_0\} \cup \{t \hookrightarrow m_{n+1} \mid t \stackrel{\text{let}}{=} \cdots \hookrightarrow m_n \in X,\ m_n \hookrightarrow m_{n+1}\}$$

where $Machine = Edge \times State$, the program points $Edge = Lab \times Lab$ are the set of edges between two program labels, and m_0 is the initial machine state. Labels are uniquely assigned to all the expressions and commands of the input C program.

We approximate the collecting semantics by $T \in Edge \rightarrow \widehat{State}$ that maps each program point to an abstract state

$$\widehat{State} = \widehat{Stack} \times \widehat{Mem} \times \widehat{Dump}$$

The abstract state at each program point approximates all the states occurring at the point in all the concrete transition sequences. The map is defined as the least fixed point of the following function:

$$\widehat{\mathcal{F}} : (Edge \rightarrow \widehat{State}) \rightarrow (Edge \rightarrow \widehat{State})$$
$$\widehat{\mathcal{F}}(T) = \lambda \langle l, l' \rangle .s \ \text{ where } \langle l,\ \bigsqcup\{s' | p \in pred(l), T\langle p, l\rangle = s'\}\rangle \hookrightarrow^{\#} \langle l', s\rangle$$

where $pred(l)$ is the set of predecessors of label l in the transition sequences and $\hookrightarrow^{\#}$ is an abstraction of the concrete transition relation $\hookrightarrow$.

2.2 Fixpoint Algorithm

The fixpoint algorithm is a chaotic working set algorithm. The working set consists of labels of expressions whose abstract state has to be re-computed. When a computed machine state for $T\langle l, l'\rangle$ is changed, we add l' to the working set, to re-compute the states of the edges from l'. The working set is a stack, hence each abstract transition step follows the program's execution flow in a depth-first order of the flow graph. When the next program points to evaluate are multiple (as when we compute conditional expressions), those two points are grouped together and pushed as a single unit to the working set stack. This grouping adds a flavor of breadth-first traversal of the flow graph. The working set algorithm selectively applies the widening and narrowing operations at the heads of flow cycles.

2.3 Accuracy Improvement

We use some techniques to improve the analysis accuracy: 1) we use widening and narrowing for interval domain[5]; 2) we use destructive assignment to achieve flow sensitive analysis except for within cyclic call chains; 3) we use context pruning to confine interval values; 4) we use function-inlining for polyvariant analysis; 5) we use static loop unrolling. Though each technique is independent of others, using all the techniques in combination results in a synergy for improving the analysis accuracy.

Table 1. Experiment result of cost reduction techniques

Version	Time[a](s)	Speed-Up	Time[b](s)	Speed-Up
none	18317.55	0%	16253.18	0%
selective join	16055.58	12.35%	14286.72	12.1%
wait-at-join	19317.67	-5.45%	13153.43	19.98%
stack obviation	3717.06	79.71%	3247.79	81.02%
all	3461.57	81.11%	2320.58	85.73%

[a] the sum of analysis time for 43 Linux kernel programs.
[b] same as *a* except one program that *wait-at-join* has bad influence upon work list algorithm.

2.4 Cost Reduction

We used three techniques for cost reduction of Airac. They are *stack obviation*, *selective join* and *wait-at-join*. From experiment results on parts of the Linux kernel, we could observe that *stack obviation* is a very powerful technique for cost reduction. The *wait-at-join* technique works well for most programs with some exceptions.

Stack Obviation. Comparing($\sqsubseteq$) and joining($\sqcup$) abstract states, which take most of the analysis time, involves applying the operations to the abstract stack component. If the abstract stack component is always reflected in other components of the machine states then we can skip applying the operations to the stack component.

Before analysis begins, Airac transforms the input program to have all stack variations of each transition be reflected on the memory. For example, conditional expression

```
(x > 0) ? 1 : 2
```

is transformed to

```
{ var tmp;  if (x > 0) tmp = 1; else tmp = 2;  tmp; }
```

Note that the original expression's branches are to push different values to the stack component. Hence estimating the final value must join the stack components from the branches. On the other hand, for the transformed expression, we don't have to consider joining the stack components because the state difference of the two branches are to be reflected by the memory component because of the assignments to the temporary variable `tmp`. This transformation costs only one more location in the abstract memory, while it avoids scanning the abstract stack component. This technique reduced our analysis time by 79.71%.

Selective Memory Join. Airac keeps track of information that indicates changed entries in abstract memory. Join operation is applied only to those changed values. Comparing with pre-state, Airac reduces size of the information by removing unchanged entries. This technique, which was also mentioned in [4], reduced our analysis time by 12.35%.

Wait-at-Join. For program points where many control-flows join, Airac delays the computation for current point until all computations for the incoming edges are done. By this, Airac can reduce redundant computations after the junction point. However, it is very costly to decide whether all threads have reached current point or not. So Airac chooses a simple strategy which waits until the working stack becomes empty. This technique is very effective for C programs that have a many junction points, e.g., large switch statements. This technique reduced our analysis time by 19.98% for most programs.

2.5 Airac's Cost, Accuracy and Scalability

We implemented Airac using nML[2] and analyzed various softwares from toy C programs to serious ones such as GNU softwares, Linux kernel sources and commercial softwares. All these commercial softwares are embedded softwares[3]. Airac found some fatal bugs in these softwares which were under development. Table 2.5 shows the result of our experiment.

"#Lines" is the number of lines of the C source files before preprocessing them. "Time" is the user CPU time in seconds. "#Buffers" is the number of buffers that may be overrun. "#Accesses" is the number of buffer-access expressions that may overrun. "#Real Bugs" is the number of buffer accesses that are confirmed to be able to cause real overruns. Two graphs in Figure 1 show Airac's scalability behavior. X axis is the size (number of lines) of the input program to analyze and Y axis is the analysis time in seconds. (b) is a microscopic view of (a)'s lower left corner. Experiment was done on a Linux system with a single Pentium4 3.2GHz CPU and 4GB of RAM.

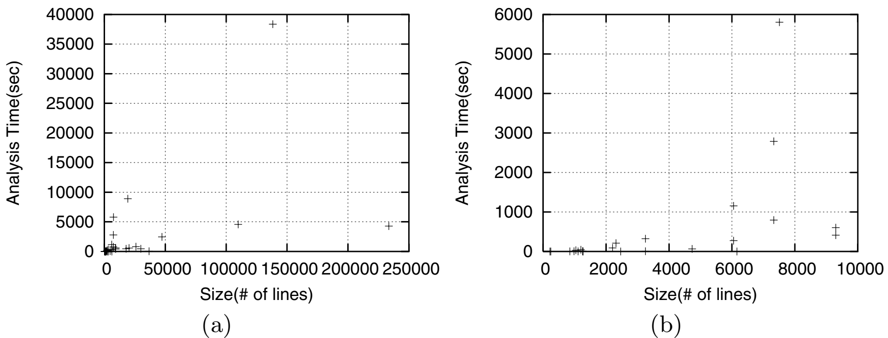


Fig. 1. Airac's scalability

Airac is scalable enough to analyze real world softwares. Airac can analyze programs of up to about 10,000 lines at once. GNU softwares such as grep, gzip and sed were analyzed as a whole. And these analyses took less than an hour to finish.

[2] Korean dialect of ML programming language. `http://ropas.snu.ac.kr/n`

[3] Their real names cannot be disclosed due to the contract.

Table 2. Analysis speed and accuracy of Airac

	Software	#Lines	Time (sec)	#Airac Alarms #Buffers	#Airac Alarms #Accesses	#Real bugs
GNU S/W	tar-1.13	20,258	576.79s	24	66	1
	bison-1.875	25,907	809.35s	28	50	0
	sed-4.0.8	6,053	1154.32s	7	29	0
	gzip-1.2.4a	7,327	794.31s	9	17	0
	grep-2.5.1	9,297	603.58s	2	2	0
Linux kernel version 2.6.4	vmax302.c	246	0.28s	1	1	1
	xfrm_user.c	1,201	45.07s	2	2	1
	usb-midi.c	2,206	91.32s	2	10	4
	atkbd.c	811	1.99s	2	2	2
	keyboard.c	1,256	3.36s	2	2	1
	af_inet.c	1,273	1.17s	1	1	1
	eata_pio.c	984	7.50s	3	3	1
	cdc-acm.c	849	3.98s	1	3	3
	ip6_output.c	1,110	1.53s	0	0	0
	mptbase.c	6,158	0.79s	1	1	1
	aty128fb.c	2,466	0.32s	1	1	1
Commercial Softwares	software 1	109,878	4525.02s	16	64	1
	software 2	17,885	463.60s	8	18	9
	software 3	3,254	5.94s	17	57	0
	software 4	29,972	457.38s	10	140	112
	software 5	19,263	8912.86s	7	100	3
	software 6	36,731	43.65s	11	48	4
	software 7	138,305	38328.88s	34	147	47
	software 8	233,536	4285.13s	28	162	6
	software 9	47,268	2458.03s	25	273	1

3 Statistical Taming of False Alarms

Reducing the number of false alarms is the key issue in sound analyses. Sound analyzers that cannot assume a particular style for the input programs can often have a high false-alarm rate. Controlling the abstraction level of the analysis will work but not very effectively. It is clear that by using less abstract domains we can distinguish more concrete values, but practically, relying solely on this approach will soon hit an unacceptable cost. Furthermore, if the analyzer must handle unlimited set of input programs, there can always be some programs that fool the analyzer. User annotation in source codes can be a powerful method [7] yet is always less desirable than being fully automatic. And, that the analyzer blindly repair its accuracy based on the annotations makes the approach vulnerable to annotation bugs. Heuristics can be applied to classify alarms into true and false ones[10]. However, unless heuristics have a strong basis, we can hardly be confident with their classifications. Heuristics can be used even during the analysis itself and may tempt us to give up the soundness and claim that such sacrifice is inevitable in order to increase the analysis precision. But, if possi-

ble with a comparable price, it is always better to know all possible bugs than knowing only some of them.

Without giving up its soundness, Airac handles the inevitable false alarms by using statistical post analysis built on top of a firm theoretical basis. The statistical analysis, given the analysis results (alarms), estimates the probability of each alarm being true. Only the alarms that have true-alarm probabilities higher than a threshold are reported to the user. Though the statistical analysis phase still has the risk of sifting out true alarms, it can reduce the risk at the user's desire. Because the underlying analyzer is sound, if the user is willing to, (s)he can receive a report that contain all the real alarms.

3.1 Bayesian Analysis

We use Bayesian statistics[8] to compute the probability of an alarm being true. Let $\oplus$ denote the event an alarm raised is true, and let $\ominus$ denote it is false. S_i denotes that a single symptom is observed in the raised alarm and $\boldsymbol{S}$ is a vector of such symptoms. The set of symptoms that we used for Airac will be presented in 3.4. $P(E)$ denotes the probability of an event E, and $P(A \mid B)$ is the conditional probability of A given B. We call the probability $P(\oplus \mid \boldsymbol{S})$ of an alarm being true given its symptoms as *the trueness of the alarm*.

Bayes' theorem is used to predict the probability of a new event from prior knowledge. To set up such knowledge base we classify alarms into true and false manually and count occurrences of each symptom in true and false alarms respectively. From this knowledge we are able to compute the trueness of new alarms using their symptoms. Using Bayes' theorem, the trueness $P(\oplus \mid \boldsymbol{S})$ can be computed as the following:

$$P(\oplus \mid \boldsymbol{S}) = \frac{P(\boldsymbol{S} \mid \oplus)P(\oplus)}{P(\boldsymbol{S})} = \frac{P(\boldsymbol{S} \mid \oplus)P(\oplus)}{P(\boldsymbol{S} \mid \oplus)P(\oplus) + P(\boldsymbol{S} \mid \ominus)P(\ominus)}.$$

By assuming each symptom in $\boldsymbol{S}$ occurs independently under each class, we have

$$P(\boldsymbol{S} \mid c) = \prod_{S_i \in \boldsymbol{S}} P(S_i \mid c) \text{ where } c \in \{\oplus, \ominus\}.$$

Here, $P(S_i \mid \oplus)$ is estimated by $\hat{\psi}$ using Bayesian analysis of our empirical data. We assume prior distributions are uniform on $[0, 1]$. Let p be the estimator of the ratio $P(\oplus)$ of true alarms to all raised alarms. Each $P(S_i \mid \oplus)$ and $P(S_i \mid \ominus)$ is estimated by θ_i and η_i respectively. Assuming that each S_i are independent in each class, the posterior distribution of $P(\oplus \mid \boldsymbol{S})$ taking our empirical data into account is established as following:

$$\hat{\psi}_j = \frac{(\prod_{S_i \in \boldsymbol{S}} \theta_i) \cdot p}{(\prod_{S_i \in \boldsymbol{S}} \theta_i) \cdot p + (\prod_{S_i \in \boldsymbol{S}} \eta_i) \cdot (1-p)} \tag{1}$$

where p, θ_i and η_i have beta distributions as

$$\begin{aligned} p &\sim Beta(N(\oplus) + 1, n - N(\oplus) + 1) \\ \theta_i &\sim Beta(N(\oplus,\ S_i) + 1, N(\oplus, \neg S_i) + 1) \\ \eta_i &\sim Beta(N(\ominus, S_i) + 1, N(\ominus, \neg S_i) + 1) \end{aligned}$$

and $N(E)$ is the number of events E counted from our empirical data. Now the estimation of p, θ_i,η_i are done by Monte Carlo method. We randomly generate $p_i, \theta_{ij}, \eta_{ij}$ values N times from the beta distributions and compute N instances of ψ_j. Then the $100(1-2\alpha)\%$ credible set of $\hat{\psi}$ is $(\psi_{j_{\alpha\cdot N}}, \psi_{j_{(1-\alpha)\cdot N}})$ where $\psi_{j_1} < \psi_{j_2} < \cdots < \psi_{j_N}$. We take the upper bound $\psi_{j_{(1-\alpha)\cdot N}}$ for $\hat{\psi}$, since the maximal probability being true is our concern as seen later.

3.2 Sifting Out False Alarms

We can use the estimated trueness for sifting out false alarms systematically. To decide whether we should sift out an alarm or not, we need a threshold to compare with the estimated $\hat{\psi}$ with $100(1-2\alpha)\%$ credibility. To choose a reasonable threshold, the user supplies two parameters defining the magnitude of risk: r_m for missing true alarms and r_f for reporting false alarms. Only their ratio, not their absolute values matter.

	$\oplus$	$\ominus$
risk of reporting	0	r_f
risk of not reporting	r_m	0

Given an alarm whose trueness is ψ, the expectation of risk when we raise an alarm is $r_f \cdot (1-\psi)$, and $r_m \cdot \psi$ when we don't. To minimize the risk, we must choose the smaller side. Hence, the threshold of trueness to report the alarm can be chosen as:

$$r_m \cdot \psi \geq r_f \cdot (1-\psi) \quad \iff \quad \psi \geq \frac{r_f}{r_m + r_f}.$$

If the trueness of an alarm can be greater than or equal to such threshold, i.e. if the upper bound of trueness $\hat{\psi}$ is greater than such threshold, then the alarm should be raised with $100(1-2\alpha)\%$ credibility. For example, the user can supply $r_m = 3, r_f = 1$ if he or she believes that not alarming for true errors have risk 3 times greater than raising false alarms. Then the threshold for the probability being true to report becomes $1/4 = 0.25$ and whenever the estimated trueness of an alarm is greater than 0.25, we should report it.

We have done some experiments with our samples of programs and alarms. Some parts of the Linux kernel and programs that demonstrate classical algorithms were used for the experiment. For a single experiment, samples were first divided into learning set and testing set. 50% of the alarms were randomly selected as learning set, and the others for testing set. Each symptom in the learning set were counted according to whether the alarm was true or false. With these pre-calculated numbers, $\hat{\psi}$ for each alarm in the testing set was estimated using the 90% credible set constructed by Monte Carlo method. Using Equation (1), we computed 2000 ψ_j's from 2000 p's and θ_i's and η_i's, all randomly generated from their distributions. We can view alarms in the testing set as alarms from new programs, since their symptoms didn't contribute to the numbers used for the estimation of $\hat{\psi}$.

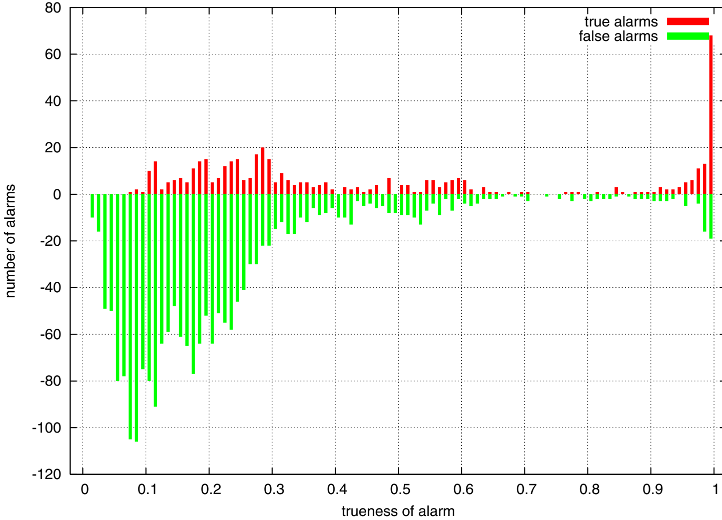


Fig. 2. Frequency of trueness in true and false alarms. False alarms are counted in negative numbers. 74.83% of false alarms have trueness less than 0.25.

The histogram in Figure 2 was constructed from the data generated by repeating the experiment 15 times. Dark bars indicate true alarms and lighter ones are false. 74.83% ($\approx$ 1504/2010) of false alarms have trueness less than 0.25, so that they can be sifted out. For users who consider the risk of missing true error is 3 times greater than false alarming, almost three quarters of false alarms could be sifted out, or preferably just deferred.

For a sound analysis, it is considered much riskier to miss a true alarm than to report a false one, so it is recommended to choose the two risk values $r_m \gg r_f$ to keep more soundness. For the experiment result Figure 2 presents, 31.40% ($\approx$ 146/465) of true alarms had trueness less than or equal to 0.25, and were also sifted out with false alarms. Although we do not miss any true alarm by lowering the threshold down to 0.07 ($r_m/r_f \approx 13$) for this case, it does not guarantee any kind of soundness in general. However, to obtain a sound analysis result, one can always set $r_f = 0$, i.e. allowing none of the alarms to be sifted out.

3.3 Ranking False Alarms

We can rank alarms by their trueness to give effective results to the user. This ranking can be used both with and without the previous sifting-out technique. By ordering alarms, we let the user handle more probable errors first. Although the trueness of true alarms are scattered over 0 through 1, we can see that most of the false alarms have small trueness. Hence, sorting by trueness and showing in decreasing order will effectively give true alarms first to the user. Figure 3 shows the cumulative percentage of observed alarms starting from trueness 1 and down using the same data in Figure 2. When the user inspects alarms having

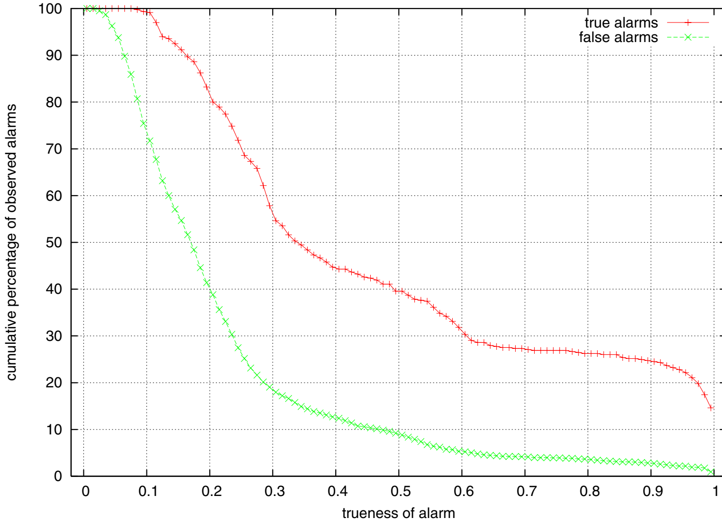


Fig. 3. Cumulative percentage of observed alarms starting from trueness 1 and down. Only 15.17% of false alarms get mixed up until 50% of the trues are observed.

high trueness first, only 15.17% (=305/2010) of false alarms gets mixed up until 50% of the true alarms are observed, where the trueness equals 0.3357.

3.4 Symptoms

We use both syntactic and semantic symptoms that may influence the analysis accuracy. The symptoms can be classified into three types: 1) syntactic context of the alarmed expressions; 2) general factors that influence the analysis accuracy; and 3) properties of the analysis results(estimated array indices).

Syntactic Context. Syntactic symptoms describe the syntactic context around the alarmed expressions. For a given alarmed expression we gather the following symptoms from the function body that contains the alarmed expression:

AfterLoop	AfterBranch
AfterReturn	InNestedLoopBody$N$
InNestedBranchBody$N$	InLoopCond
InBranchCond	InFunParam
InNestedFunParam	InRightOfAnd

AfterLoop and AfterBranch are respectively turned on when loops and branches appear before the alarmed expressions. These symptoms are for false alarms; loop and branch can decrease analysis accuracy due to the join operations at their flow-join points. AfterReturn is on when a return statement precedes the alarmed expression. This symptom is for false alarms; while the return statements in the middle of function body are often for exiting on erroneous cases

static analysis can blindly propagate such erroneous cases to independent yet later array access expressions. InNestedLoopBody$N$ and InNestedBranchBody$N$ are for true alarms because programmer are easy to mistake inside nested loops and branches. Since we found that simple nested structures were common in both true and false alarms we refined symptoms by their nesting depth $N = 1, 2, 3$, or > 3. InLoopCond and InBranchCond are on when alarms are inside the condition expressions, and InFuncParam and InNestedFuncParam are on when alarms are inside function's actual argument expressions. These four symptoms are for true alarms because it is likely that expressions in those contexts are more carefully checked by programmers than expressions in other contexts. InRightOfAnd is for alarms in the right hand side of the logical-and `&&` operator. This symptom is for false alarms because C's short-circuit semantics can skip executing the `&&` operator's rhs expressions.

General Accuracy Factors. Symptoms that can be detected only during the analysis can be useful indicators. Following symptoms are collected during the fixed point iterations:

Join$N$	Pruned	
Narrowed	PassedVal	InStructure

The number of program points where the join operation occurs affects the analysis accuracy. This situation is captured by symptom Join$N$. N is the number of such program points before an alarmed expression. N ranges over $\{1, \ldots, 10, > 10\}$. The context pruning and the narrowing operations too are influencing factors for the analysis accuracy. Pruned and Narrowed are on when those operations are successful. PassedValue is on when array index values are passed as arguments, and InStructure is on when the target arrays are pointed to from some data structures (e.g. record fields). These two symptoms are for true alarms because such complicated use of the target arrays and indices are likely to be confused.

Properties of Estimated Array Indices. The analysis results themselves are used as symptoms too:

TopIndex HalfInfiniteIndex FiniteIndex

If an estimated array index is the whole integer interval it is likely to be a false alarm (TopIndex). HalfInfiniteIndex is on when an index interval is half-infinite like $[1, \infty]$. Conversely, array indices with exact boundaries(FiniteIndex) strongly suggest true alarms.

4 Related Work

Reducing false alarms has always been a critical problem in static analysis. Existing tools have addressed the false alarm problem by 1) giving up the soundness

of analysis (e.g. SPLINT[14], ARCHER[13]); 2) depending on user annotations (e.g. CSSV[7], SPLINT[14]); 3) limiting the target programs (e.g. ASTRÉE[4,6]); 4) heuristically classifying the alarms into either true ones or false ones (e.g. Z-ranking[10]).

Airac differs from existing tools in that it uses Bayesian statistical analysis to classify the alarms by their probabilities being true. Our Bayesian approach can be orthogonally used with the user annotation approach. As of the analysis itself, it is sound, does not rely on user annotations, covers the full set of ANSI C constructs, and scale up to several 10K LOC.

CSSV[7] and SPLINT[14] rely on user annotations to reduce the false alarms. With imprecise or null user annotation, these tools have rapidly increasing false-alarm rate. ARCHER[13] is not sound, having a low detection-rate for bugs. ASTRÉE[4,6] is a static program analyzer aiming at verifying the absence of run time errors in a limited number of avionics controller programs in C. This analyzer reports zero or very few false alarm. It excludes several C features (e.g. union types, dynamic memory allocation, and unbounded recursive function calls).

The one most directly related to our Bayesian approach is Z-ranking[10]. It ranks alarms by heuristics. It first partitions successes (e.g. safe buffer accesses) and failures (e.g. buffer overrun alarms) into groups. In each group, using a three heuristics, it computes "z-score" for each alarm being true. Alarms in decreasing order of z-scores are presented to the user. This approach has two drawbacks. The heuristics are only about the relative numbers of successes and failures in each group and they do not mention about any systematic method on how to partition the alarms. Thus if the partitioning happen to group alarms about which the heuristics fail to reflect the reality, Z-ranking can be ineffective. In comparison, our statistical approach is more robust. Our method has no arbitrary parameter like the "partitioning" in Z-ranking; it's competence does not rely on a particular factor of the method because the set of symptoms, which correspond to our method's heuristics, are extensive covering both the analyzer's internal behaviors and the input programs' syntactic characteristics; and lastly, thanks to the Bayesian framework's learning capability, our method's competence will improve as the analysis results are accumulated.

5 Conclusion and Discussion

We present that combining, in a realistic setting, a domain-unaware static analyzer with a Bayesian analysis can be a viable approach to handle false alarms. Our analyzer Airac, which collects all the true buffer-overrun points in ANSI C programs, is sound and its cost-accuracy improvement is achieved by techniques that static analysis community has long accumulated. For still inevitable false alarms we design a Bayesian post analysis. The statistical analysis, given the analysis results (alarms), estimates the probability of each alarm being true. The probabilities are used to sift out probable false alarms and prioritize true alarms. Only the alarms that have trueness higher than a threshold are reported

to the user, and the alarms are sorted by the probability before reporting, so that the user can check highly probable errors first. In our experiments with Linux kernel sources and some textbook programs, if the user set the risk of missing true error is about 3 times greater than false alarming, 74.83% of false alarms could be filtered; and only 15.17% of false alarms were mixed up until the user observes 50% of the true alarms.

The Bayesian analysis' competence heavily depends on how we define symptoms. Since the inference framework is known to work well, better symptoms and feasible size of pre-classified alarms is the key of this approach. We think promising symptoms are tightly coupled with analysis' weakness and/or its preciseness, and some fair insight into the analysis is required to define them. However, since general symptoms, such as syntactic ones, are tend to reflect the programming style, and such patterns are well practiced within organizations, we believe local construction and use of the knowledge base of such simple symptoms will still be effective. Furthermore, we see this approach easily adaptable to possibly any kind of static analysis.

Another approach to handling false alarms is to equip the analyzer with all possible techniques for accuracy improvement and let the user choose a right combination of the techniques for her/his programs to analyze. The library of techniques must be extensive enough to specialize the analyzer for as wide spectrum of the input programs as possible. This approach lets the user decide how to control false alarms, while our Bayesian approach lets the analysis designer decide by choosing the symptoms based on the knowledge about the weakness and strength of his/her analyzer. We see no reason we cannot combine the two approaches.

Acknowledgements. We thank Jaeyong Lee for helping us design our Bayesian analysis, Hakjoo Oh and Yikwon Hwang for collecting and identifying false alarm cases, and anonymous referees for helpful comments.

References

1. bugtraq. www.securityfocus.com.
2. CERT/CC advisories. www.cert.org/advisories.
3. James O. Berger. *Statistical Decision Theory and Bayesian Analysis, 2nd Edition.* Springer, 1985.
4. Bruno Blanchet, Patric Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antonie Mine, David Monnizux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–207, June 2003.
5. Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP '92: Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295. Springer-Verlag, 1992.

6. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrée analyzer. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer-Verlag, 2005.
7. Nurit Dor, Michael Rodeh, and Mooly Sagiv. 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*, pages 155–167. ACM Press, 2003.
8. Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis.* Text in Statistical Science. Chapman & Hall/CRC, second edition edition, 2004.
9. David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.
10. Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In Radhia Cousot, editor, *SAS '03: Proceedings of the 10th Annual International Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 295–315. Springer, 2003.
11. Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.
12. N. Metropolis and S. Ulam. The Monte Carlo method. *Journal of the American Statistical Association*, 44(247):335–341, September 1949.
13. Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM Press, 2003.
14. Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106. ACM Press, 2004.

Banshee: A Scalable Constraint-Based Analysis Toolkit*

John Kodumal[1] and Alex Aiken[2]

[1] EECS Department, University of California, Berkeley
[2] Computer Science Department, Stanford University

Abstract. We introduce BANSHEE, a toolkit for constructing constraint-based analyses. BANSHEE's novel features include a code generator for creating customized constraint resolution engines, incremental analysis based on backtracking, and fast persistence. These features make BANSHEE useful as a foundation for production program analyses.

1 Introduction

Program analyses that are simultaneously scalable, accurate, and efficient remain expensive to develop. One approach to lowering implementation cost is to express the analysis using *constraints*. Constraints separate analysis *specification* (constraint generation) from analysis *implementation* (constraint resolution). By exploiting this separation, designers can benefit from existing algorithms for constraint resolution. This separation helps, but leaves several problems unaddressed. A generic constraint resolution implementation with no knowledge of the client may pay a large performance penalty for generality. For example, the fastest hand-written version of Andersen's analysis [12] is much faster than the fastest version built using a generic toolkit [2]. Furthermore, real build systems require separate analysis to fit with separate compilation. Small edits to projects are the norm; reanalyzing an entire project for each small change is unrealistic.

We have built BANSHEE, a constraint-based analysis toolkit that addresses these problems [14]. BANSHEE succeeds BANE, our first generation toolkit for constraint-based program analysis [2]. BANSHEE inherits several features from BANE, particularly support for *mixed constraints*, which allow several constraint formalisms to be combined in one application (Section 2). BANSHEE also provides a number of innovations over BANE that make it more useful and easier to use:

– We use a code generator to specialize the constraint back-end for each program analysis (Section 3). The analysis designer describes a set of constructor signatures in a specification file, which BANSHEE compiles into a specialized constraint resolution engine. Specialization allows checking a host of correctness conditions statically. Software maintenance also improves: specialization

* This research was supported in part by NASA Grant No. NNA04CI57A; NSF Grant Nos. CCR-0234689, CCR-0085949, and CCR-0326577. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 218–234, 2005.
© Springer-Verlag Berlin Heidelberg 2005

allows the designer to modify the specification without wholesale changes to the handwritten part of the analysis. Finally, BANSHEE is truly modular; new constraint formalisms (or *sorts*) can be added without changing existing sorts.
- We have added support for a limited form of incremental analysis via *backtracking*, which allows constraint systems[1] to be rolled back to any previous state (Section 4). Backtracking can be used to analyze large projects incrementally: when a source file is modified, we roll back the constraint system to the state just before the analysis of that file. By choosing the order in which files are analyzed to exploit locality among file edits, we show experimentally that backtracking is very effective in avoiding reanalysis of files.
- We have added support for efficient serialization and deserialization of constraint systems (Section 5). The ability to save and load constraint systems is important for integrating BANSHEE-derived analysis into real build processes as well as for supporting incremental analysis. This feature is nontrivial, especially in conjunction with backtracking; our solution exploits BANSHEE's use of explicit regions for memory management.
- We have written BANSHEE from the ground up, implementing all important optimizations in BANE, while the code generation framework has enabled us to add a host of engineering and algorithmic improvements. In a case study, we show how BANSHEE's specification mechanism allows various points-to analyses to be easily expressed (Section 6) while the performance is nearly 100 times faster than BANE on some standard benchmarks (Section 7).

BANSHEE has reached the point of being a productive tool for developing experimental and production-quality program analyses. As evidence, we cite several BANSHEE applications. A BANSHEE-based polyvariant binding-time analysis for partial evaluation of graphics programs has been used in production at a major effects studio [19,20]. BANSHEE has been used as part of a software updateability analysis tool [27]. A BANSHEE-based type inference system for Prolog has been developed [23]. Also, for two years a BANSHEE pointer analysis was used as a prototype global alias analysis in a development branch of the `gcc` compiler.

2 Mixed Constraints

BANSHEE is built on *mixed constraints*, which allow multiple constraint *sorts* in one application. A sort s is a tuple $(V_\mathsf{s}, C_\mathsf{s}, O_\mathsf{s}, R_\mathsf{s})$ where V_s is a set of variables, C_s is a set of constructors, O_s is a set of operations, and R_s is a set of constraint relations. Each n-ary constructor $c_\mathsf{s} \in C_\mathsf{s}$ and operation $op_\mathsf{s} \in O_\mathsf{s}$ has a signature $\iota_1 \dots \iota_n \to \mathsf{s}$ where ι_i is either s_i or $\overline{\mathsf{s}}_i$ for sorts s_i. Overlined arguments in a signature are contravariant; all other arguments are covariant. A n-ary constructor c_s is *pure* if the sort of each of its arguments is s. Otherwise, c_s is *mixed*. For

[1] Note that BANSHEE's solvers are all online, so existing constraints are maintained in a partially solved form as new constraints are added.

$$\begin{array}{ll}
\mathcal{C} \wedge \{\mathcal{X} \subseteq_{\mathsf{Set}} \mathcal{X}\} \rightarrow \mathcal{C} & \mathcal{C} \wedge \{c(\ldots) \subseteq_{\mathsf{Set}} d(\ldots)\} \rightarrow \text{inconsistent} \\
\mathcal{C} \wedge \{e_{\mathsf{Set}} \subseteq_{\mathsf{Set}} 1\} \rightarrow \mathcal{C} & \mathcal{C} \wedge \{c(\ldots) \subseteq_{\mathsf{Set}} 0\} \rightarrow \text{inconsistent} \\
\mathcal{C} \wedge \{0 \subseteq_{\mathsf{Set}} e_{\mathsf{Set}}\} \rightarrow \mathcal{C} & \mathcal{C} \wedge \{1 \subseteq_{\mathsf{Set}} c(\ldots)\} \rightarrow \text{inconsistent} \\
 & \mathcal{C} \wedge \{1 \subseteq_{\mathsf{Set}} 0\} \rightarrow \text{inconsistent}
\end{array}$$

$$\mathcal{C} \wedge \{c(e_{\mathsf{s}_1}, \ldots, e_{\mathsf{s}_n}) \subseteq_{\mathsf{Set}} c(e'_{\mathsf{s}_1}, \ldots, e'_{\mathsf{s}_n})\} \rightarrow \mathcal{C} \wedge \bigwedge_i \begin{cases} \{e_{\mathsf{s}_i} \subseteq_{s_i} e'_{\mathsf{s}_i}\} & c \text{ covariant in } i \\ \{e'_{\mathsf{s}_i} \subseteq_{s_i} e_{\mathsf{s}_i}\} & c \text{ contravariant in } i \end{cases}$$

Fig. 1. Constraint resolution for the Set sort

a sort s, a set of variables, constructors, and operations defines a language of s-expressions e_{s}:

$$\begin{array}{lll}
e_{\mathsf{s}} ::= & & \\
| & v & v \in V_{\mathsf{s}} \\
| & c_{\mathsf{s}}(e_{\mathsf{s}_1}, \ldots, e_{\mathsf{s}_n}) & c_{\mathsf{s}} \text{ with signature } \iota_1 \ldots \iota_n \rightarrow \mathsf{s} \\
 & & \text{and } \iota_i \text{ is } \mathsf{s}_i \text{ or } \overline{\mathsf{s}}_i \\
| & op_{\mathsf{s}}(e_{\mathsf{s}_1}, \ldots, e_{\mathsf{s}_n}) & op_{\mathsf{s}} \text{ with signature } \iota_1 \ldots \iota_n \rightarrow \mathsf{s} \\
 & & \text{and } \iota_i \text{ is } \mathsf{s}_i \text{ or } \overline{\mathsf{s}}_i
\end{array}$$

Constraints between expressions are written $e_{1_{\mathsf{s}}}\ r_{\mathsf{s}}\ e_{2_{\mathsf{s}}}$ where r_{s} is a constraint relation ($r_{\mathsf{s}} \in R_{\mathsf{s}}$). Each sort s has two distinguished constraint relations: an inclusion relation (denoted $\subseteq_{\mathsf{s}}$) and a unification relation (denoted $=_{\mathsf{s}}$). A constraint system $\mathcal{C}$ is a finite conjunction of constraints.

To fix ideas, we introduce two Banshee sorts and informally explain their semantics. A formal presentation of the semantics of mixed constraints is given in [8]. We leave the set of constructors C_{s} unspecified in each example, as this set parameterizes the constraint language and is application-specific.

Example 1. *The Set sort is the tuple:* $(V_{\mathsf{Set}}, C_{\mathsf{Set}}, \{\cup, \cap, 0, 1\}, \{\subseteq, =\})$.

Here V_{Set} is a set of set-valued variables, $\cup, \cap$, $\subseteq$, and $=$ are the standard set operations, 0 is the empty set, and 1 is the universal set. Each pure Set expression denotes a set of *ground terms*: a constant or a constructor $c_{\mathsf{Set}}(t_1, \ldots, t_n)$ where each t_i is a ground term. A subset of the resolution rules for the Set sort is shown in Figure 1; Banshee implements these as left-to-right rewrite rules.

Example 2. *The Term sort is the tuple:* $(V_{\mathsf{Term}}, C_{\mathsf{Term}}, \{0, 1\}, \{\leq, =\})$.

Here V_{Term} is a set of term-valued variables, and $=$ and $\leq$ are unification and conditional unification [25], respectively. The meaning of a pure Term expression is, as expected, a constant or a constructor $c_{\mathsf{Term}}(t_1, \ldots, t_n)$ where t_i are terms. A subset of the resolution rules for the Term sort is shown in Figure 2.[2]

A system of mixed constraints defines a directed graph where nodes are expressions and edges are *atomic constraints* between expressions. A constraint

[2] We use "term" to mean both a sort and ground terms (trees) built by constructor application. The intended meaning should be clear from context.

$$\begin{array}{rll}
\mathcal{C} \wedge \{\mathcal{X} =_{\mathsf{Term}} \mathcal{X}\} & \rightarrow \mathcal{C} & \\
\mathcal{C} \wedge \{c(e_{\mathsf{s}_1}, \ldots, e_{\mathsf{s}_n}) =_{\mathsf{Term}} c(e'_{\mathsf{s}_1}, \ldots, e'_{\mathsf{s}_n})\} & \rightarrow \mathcal{C} \wedge \bigwedge_i \{e_{\mathsf{s}_i} =_{\mathsf{s}_i} e'_{\mathsf{s}_i}\} & \\
\mathcal{C} \wedge \{e_{\mathsf{Term}} \leq e'_{\mathsf{Term}}\} & \rightarrow \mathcal{C} \wedge \{e_{\mathsf{Term}} = e'_{\mathsf{Term}}\} & \text{if } e_{\mathsf{Term}} \text{ is not } 0 \\
& \mathcal{C} & \text{if } e_{\mathsf{Term}} \text{ is } 0 \\
\mathcal{C} \wedge \{c(\ldots) =_{\mathsf{Term}} d(\ldots)\} & \rightarrow \text{inconsistent} &
\end{array}$$

Fig. 2. Constraint resolution for the Term sort

is atomic if the left- or right-hand side is a variable. To solve the constraints, the constraint graph is closed under the resolution rules for each sort as well as a transitive closure rule.

3 Specialization

This section describes the compilation strategy used in Banshee. We omit the low-level details, which are straightforward, and focus on explaining the advantages of our approach.

To use Banshee, the analysis designer writes a specification file defining the constructor signatures for the analysis. Consider a constructor *fun* modeling a function type in a unification-based type inference system with an additional set component to track the function's latent effect, in the style of a type and effect system. The signature is:

$$fun : \mathsf{Term} \ * \ \mathsf{Term} \ * \ \mathsf{Set} \rightarrow \mathsf{Term}$$

which is specified in Banshee as follows (see Section 6 for more explanation):

```
data l_type : term = fun of l_type * l_type * effect
and effect : set
```

In Bane, this signature can be declared at run-time, even during constraint resolution. Bane is an interpreter for a language of constructors and resolution rules, and as such it has the overhead of an interpreter. For example, to apply the constructor *fun*, Bane checks at run-time that there are the right number of arguments of the correct sorts. There is also interpretive overhead in constraint resolution. Consider implementing the rule for constraints between two *fun* expressions:

$$\begin{array}{l}
\mathcal{C} \wedge \{fun(e_{\mathsf{Term}_1}, e_{\mathsf{Term}_2}, e_{\mathsf{Set}}) =_{\mathsf{Term}} fun(e'_{\mathsf{Term}_1}, e'_{\mathsf{Term}_2}, e'_{\mathsf{Set}})\} \rightarrow \\
\qquad \mathcal{C} \wedge \{e_{\mathsf{Term}_1} =_{\mathsf{Term}} e'_{\mathsf{Term}_1}\} \wedge \{e_{\mathsf{Term}_2} =_{\mathsf{Term}} e'_{\mathsf{Term}_2}\} \wedge \{e_{\mathsf{Set}} =_{\mathsf{Set}} e'_{\mathsf{Set}}\}
\end{array}$$

To implement this rule, Bane uses the signature to choose the correct constraint relation and the directionality for each component. Because this work is done dynamically, both require either run-time tests or dynamic dispatch to implement.

From experience we have learned that analyses rely on a small, fixed number of constructors that can be specified statically. BANSHEE uses static signatures to implement customized versions of the constructors and the constraint resolution rules, which allows us to eliminate many kinds of dynamic checks statically. For example, consider again the signature of the *fun* constructor, now declared statically in BANSHEE. From this signature, BANSHEE generates a C function with the following prototype:

```
l_type fun(l_type e0, l_type e1, effect e2)
```

Notice that the dynamic arity and sort checks are no longer necessary—the C type system guarantees that calls to this function have the correct number of arguments (the arity check) and that the types of any actuals match the formal arguments (the sort checks). Similarly, BANSHEE can statically discharge the dynamic checks in resolution rules discussed above.

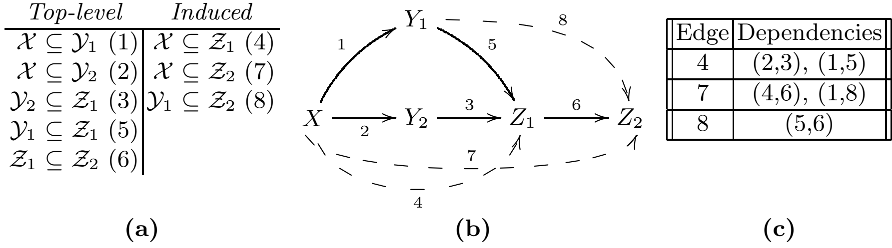
One of the most important advantages of BANSHEE specifications is that they make program analyses easier to debug and maintain. After writing a BANSHEE specification, the analysis designer's main task is to write C code to traverse abstract syntax, calling functions from the generated interface to build expressions, generate constraints, and extract solutions. This task is typically straightforward, as there should be a tight correspondence between type judgments and the BANSHEE code to implement them. Continuing with the type and effect example, we might wish to implement the following rule for function application:

$$\frac{\begin{array}{cc} \Gamma \vdash e_1 : \tau_1; \epsilon_1 & \Gamma \vdash e_2 : \tau_2; \epsilon_2 \\ \tau_1 = \tau_2 \rightarrow^{\epsilon} \alpha & \alpha, \epsilon \text{ fresh} \end{array}}{\Gamma \vdash e_1\ e_2 : \alpha; \epsilon_1 \cup \epsilon_2 \cup \epsilon} \quad \text{(App)}$$

Assuming a typical set of AST definitions, the corresponding BANSHEE code to implement this rule is[3]:

```
struct type_and_effect analyze(env gamma, ast_node n) {
  if (is_app_node(n)) {
    l_type tau1, tau2, alpha;
    effect epsilon1, epsilon2, epsilon;
    (tau1, epsilon1) = analyze(gamma,n->e1);
    (tau2, epsilon2) = analyze(gamma,n->e2);
    alpha = l_type_fresh();
    epsilon = effect_fresh();
    l_type_unify(tau1, fun(tau2,alpha,epsilon));
    return (alpha, effect_union([epsilon1;epsilon2;epsilon]));
  }
  ...
}
```

[3] We use a little syntactic sugar for pairs and lists in C to avoid showing the extra type declarations.



Top-level	*Induced*
$\mathcal{X} \subseteq \mathcal{Y}_1$ (1)	$\mathcal{X} \subseteq \mathcal{Z}_1$ (4)
$\mathcal{X} \subseteq \mathcal{Y}_2$ (2)	$\mathcal{X} \subseteq \mathcal{Z}_2$ (7)
$\mathcal{Y}_2 \subseteq \mathcal{Z}_1$ (3)	$\mathcal{Y}_1 \subseteq \mathcal{Z}_2$ (8)
$\mathcal{Y}_1 \subseteq \mathcal{Z}_1$ (5)	
$\mathcal{Z}_1 \subseteq \mathcal{Z}_2$ (6)	

Edge	Dependencies
4	(2,3), (1,5)
7	(4,6), (1,8)
8	(5,6)

Fig. 3. (a) Constraints. (b) Constraint graph. (c) Edge dependency list.

This code is representative of the "handwritten" part of a Banshee analysis. Banshee's code generator makes the handwritten part clean: there is a close correspondence between clauses in the type rules and the Banshee code to implement them.

4 Incremental Analysis

Incremental analysis is important in large projects where it is necessary to maintain a global analysis in the face of small edits. In this section we describe Banshee's support for a form of incremental analysis via *backtracking*, a mechanism that is efficient, simple to implement, and applicable to a wide variety of program analysis systems besides Banshee.

We assume *constraint additions*, *constraint deletions*, and *queries* (testing whether the constraints satisfy some fact) are arbitrarily interleaved. Additions and queries are handled by on-line solving; the trick is handling deletions.

Consider adding constraints (1)-(3) in Figure 3(a) to an initially empty constraint system. Constraints (2)-(3) cause constraint (4) to be added (by transitive closure). We say (1)-(3) are *top-level* constraints (added by the user, solid lines in the constraint graph in Figure 3(b)) and (4) is an *induced* constraint (added by the closure rules, dashed lines in Figure 3(b)).

At this point, if we delete constraint (2), then constraint (4) must be deleted as well, as it would no longer be included in the closed constraint graph. We say constraint(4) *depends on* constraints (2) and (3). The key to incremental analysis is tracking such dependency information.

A straightforward way to track precise dependency information is to explicitly maintain a list of constraints on which each induced constraint depends. For example, adding constraint (5) adds edge (4) again, so the entry (1,5) must be included in (4)'s dependency list. This approach is costly in space, because an induced constraint often is added multiple times [28]. Figures 3(b) and (c) show the closed constraint graph and edge dependencies after constraint (6) and its induced constraints are added to the graph.

Besides the space cost, another major concern is the engineering effort required to support fine-grained incrementality. To our knowledge, there is no

general, practical incremental algorithm for maintaining arbitrary data structures [7]. Adding ad-hoc support for incremental updates to each BANSHEE sort is daunting, as the algorithms are highly optimized. For example, our set constraint solver uses a union-find algorithm to implement partial online cycle elimination [9]. Adding incremental support to union-find alone is not easy—in fact, some published solutions are incorrect [10].

Instead of computing precise dependencies, we use *backtracking*, which is based on an approximation: each induced constraint depends on all constraints introduced earlier. Thus, to delete constraint c, we delete c and all induced constraints added after c. Because this notion of dependency is approximate, we must solve the resulting constraint system to rediscover induced constraints that should not have been deleted.

Backtracking is implemented by time stamping constraints. Deleting constraint i is done by scanning all constraints (edges), deleting any induced constraint with a timestamp greater than i, and solving. Consider again the example in Figure 3, and take a constraint's number to be its timestamp. We see that deleting constraint (6) also deletes constraints (7) and (8), but because both (7) and (8) only depend on (6), backtracking is as precise as tracking edge dependencies in this case. Going further and also deleting constraint (3), however, deletes induced constraint (4), which is rediscovered through the transitive path (1,5) when the resulting system is solved. While backtracking can overestimate the set of deleted constraints and incur extra work in rediscovering induced constraints, it has practical advantages over computing edge dependencies. Backtracking uses a linear scan of the constraint graph's edges, while the precise incremental algorithm is linear in the size of edge dependency lists, which may be quadratic in the number of graph edges. The storage overhead of backtracking is just a timestamp per edge, while edge dependency lists raise the worst case storage for an n-node graph from $\mathcal{O}(n^2)$ to $\mathcal{O}(n^3)$.

We have also devised a new data type called a *tracked reference* that adds efficient backtracking support to general data structures. This abstraction simplified the task of incorporating backtracking into BANSHEE, especially in the presence of optimizations like cycle elimination and projection merging [9,28]. A tracked reference is a mutable reference that maintains a repository of its old versions. Each tracked reference is tied to a clock; each tick of the clock checkpoints the reference's state. When the clock is rolled back, the previous state is restored. Rolling back is a destructive operation. Implementing tracked references is simple: it suffices to maintain a stack of the references' old contents. A backtracking operation pops entries off the stack until the last clock tick. Appendix A includes a compilable O'Caml implementation of tracked references. We have implemented backtracking by incorporating tracked references systematically into each BANSHEE data structure. Interestingly, we do not pay any cost for this factoring. For example, applying tracked references to a standard union-find algorithm yields an algorithm equivalent to a well-known algorithm for union-find with backtracking [29].

The basic approach to adding backtracking to a static analysis is as follows. Given a fully analyzed program and a program edit, we backtrack to the first constraint that changed as a result of the edit[4] and re-analyze the program from that point forward. For projects using standard version control systems, it is natural to use file granularity for changes. An interactive development environment may provide granularity, and BANSHEE itself can backtrack at constraint level granularity. Thus, we maintain a stack of analyzed files. If a file is modified, we pop the stack to that file, backtrack, and re-analyze all popped files.

Files are pushed back on to the stack in the order they are (re-)analyzed, but we have the flexibility to choose this order. We believe there is locality in program modifications: developers work on one or a few files at a time, and new code is more likely to be modified than old, stable code. When reanalyzing files, the order of files on the stack is preserved except that the modified file is analyzed last, thus placing it at the top of the stack, reflecting the belief that it is most likely to be the next file modified. As long as there is locality among edits, edited files will on average be close to the top of the stack under this strategy.

5 Persistence

We briefly explain our approach to making BANSHEE's constraint systems persistent. Persistence is useful when incorporating incremental analyses into standard build processes. We require persistence (rather than a feature to save and load in some simpler format) as we must reconstruct the representation of our data structures to support online constraint solving and backtracking.

Persistence is achieved by adding serialization and deserialization to the region-based memory management library used by BANSHEE [11]. Constraint systems are saved by serializing a collection of regions, and loaded by deserializing regions and updating pointer values stored in those regions. Initially, we implemented serialization using a standard pointer tracing approach, but found this strategy to be very slow because pointer tracing has poor spatial locality. Region-based serialization writes sequential pages of memory to disk, which is orders of magnitude faster. To handle deserialization, we associate an update function with each region, which is called on each object in the region to update any pointer-valued fields. With region-based serialization, we are able to serialize a 170 MB constraint graph in 2.4 seconds vs. 30 seconds to serialize the same graph by tracing pointers.

6 Case Study: Points-to Analysis

We continue with realistic examples derived from points-to analyses formulated in BANSHEE, showing how BANSHEE can be used to explore different design points and prototype variations of a given program analysis. In the first three examples, we refine the degree of subtyping used in the points-to analysis; much

[4] Here we also remove top-level constraints that may have changed due to the edit.

$$\frac{\Gamma(x) = ref(\ell_x, \mathcal{X}_{\ell_x}, \overline{\mathcal{X}}_{\ell_x})}{\Gamma \vdash x : ref(\ell_x, \mathcal{X}_{\ell_x}, \overline{\mathcal{X}}_{\ell_x})} \text{ (Var)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \&e : ref(0, \tau, \overline{1})} \text{ (Addr)}$$

$$\frac{\Gamma \vdash e : \tau \quad \tau \subseteq ref(1, \mathcal{T}, \overline{0})}{\Gamma \vdash *e : \mathcal{T}} \text{ (Deref)} \qquad \frac{\begin{array}{cc} \Gamma \vdash e_1 : \tau_1 & \Gamma \vdash e_2 : \tau_2 \\ \tau_1 \subseteq ref(1, 1, \overline{\mathcal{T}}_1) & \tau_2 \subseteq ref(1, \mathcal{T}_2, \overline{0}) \\ \multicolumn{2}{c}{\mathcal{T}_2 \subseteq \mathcal{T}_1} \end{array}}{\Gamma \vdash e_1 \ = \ e_2 : \tau_2} \text{ (Assign)}$$

Fig. 4. Constraint generation for Andersen's analysis

research on points-to analysis has focused on this issue [5,24,25]. In the fourth example, we extend points-to analysis to receiver class analysis in an object-oriented language with explicit pointer operations (e.g. C++). This analysis computes the function call graph on the fly instead of using a pre-computed call graph obtained from a coarser analysis (e.g., class hierarchy analysis).

6.1 Andersen's Analysis

Andersen's points-to analysis constructs a *points-to graph* from a set of abstract memory locations $\{\ell_1, \ldots, \ell_n\}$ and set variables $\mathcal{X}_{\ell_1}, \ldots, \mathcal{X}_{\ell_n}$. Intuitively, a reference is an object with an abstract location and methods $get : void \rightarrow \mathcal{X}_{\ell_x}$ and $set : \mathcal{X}_{\ell_x} \rightarrow void$, where $\mathcal{X}_{\ell_x}$ represents the points-to set of the location. Updating the location corresponds to applying the *set* function to the new value. Dereferencing a location corresponds to applying the *get* function. References are modeled by a constructor *ref* with three fields: a constant ℓ_x representing the abstract location, a covariant field $\mathcal{X}_{\ell_x}$ representing the *get* function, and a contravariant field $\overline{\mathcal{X}}_{\ell_x}$ representing the *set* function.

Figure 4 shows a subset of the inference rules for Andersen's analysis for C programs. The type judgments assign a set expression to each program expression, possibly generating some side constraints. To avoid having separate rules for l-values and r-values, each type judgment infers a set denoting an l-value. Hence, the set expression in the conclusion of (Var) denotes the location of program variable x, rather than its contents.

In BANSHEE, Andersen's analysis is specified as follows:

```
specification andersen : ANDERSEN =
  spec
    data location : set
    data T : set = ref of +location * +T * -T
  end
```

We outline BANSHEE's specification syntax, which is inspired by ML recursive data type declarations. Each `data` declaration defines a disjoint alphabet of constructors. For example, the declaration `data location : set` defines `location` to be a collection of constructors of sort Set. The `location` alphabet serves only as a source of fresh constants, modeling the statically unknown set of abstract locations. While static constructor signatures are an important idea in BANSHEE,

dynamic sets of constants are useful in many analyses. But all constants have a fixed, known signature, so generating them dynamically does not interfere with any of our static optimizations.

Each `data` declaration may be followed by an optional list of `|`-separated constructor declarations defining the (statically fixed) set of n-ary constructors. In this example, we define a single ternary constructor `ref`, which uses *variance annotations*. A signature element prefixed with `+` (resp. `-`) denotes a covariant (resp. contravariant) field. By default, fields are nonvariant.

6.2 Steensgaard's Analysis and One Level Flow

Andersen's analysis has cubic time complexity. Steensgaard's coarser, near-linear time analysis is implemented using the Term sort. The Andersen's specification is modified by eliminating the duplicate `T` field in the ref constructor and removing variance annotations:

```
specification steensgaard : STEENSGAARD =
  spec
    data location : set
    data T : term = ref of location * T
  end
```

Experience shows the lack of subtyping in Steensgaard's analysis leads to many spurious points-to relations. Another proposal is to use one level of subtyping. Restricting subtyping to one level is nearly as accurate as full subtyping [5]. Altering the specification to support the new analysis is again simple:

```
specification olf : OLF
  spec
    data location : set
    data T : set = ref of +location * T
  end
```

Recall the `location` field models a set of abstract locations. Making this field covariant allows subtyping at the top level. However, the `T` field is nonvariant, which restricts subtyping to the top level: at lower levels, the engine performs unification. An alternative explanation is that this signature implements the following sound rule for subtyping with updateable references [1]:

$$\frac{\ell_x \subseteq \ell_y \qquad \mathcal{X}_{\ell_x} = \mathcal{X}_{\ell_y}}{ref(\ell_x, \mathcal{X}_{\ell_x}) \leq ref(\ell_y, \mathcal{X}_{\ell_y})} \text{ (sub-ref)}$$

6.3 Receiver Class Analysis

Now that we have explored subtyping in points-to analysis, we focus on adding new capabilities to the analysis. We use the points-to information as the basis of a receiver class analysis (RCA) for an object-oriented language with explicit

pointer operations. RCA approximates the set of classes to which each expression in the program can evaluate. In a language like C++, the analysis must also use points-to information to track the effects of pointer operations.

In addition to modeling object initialization and pointer operations, our analysis must accurately simulate the effects of method dispatch. To accomplish these tasks, new constructors representing class definitions and dispatch tables are added to our points-to analysis specification. To simplify the example, we assume that methods have a single formal argument in addition to the implicit `this` parameter. Here is the BANSHEE specification for this example, using Andersen's analysis as our base points-to analysis:

```
specification rca : RCA =
  spec
    data location : set
    data T : set = ref of +location * +T * -T
                 | class of +location * +dispatch
    and dispatch : row(method)
    and method : set = fun of -T * -T * +T
  end
```

The `class` constructor contains a location field containing the name of the class and a `dispatch` field representing the dispatch table for that class' objects. Notice that `dispatch` uses a new sort, Row [8]. We first explain how this abstraction is intended to work before describing the Row sort.

An object's dispatch table is modeled as a collection of methods (each in turn modeled by the `method` constructor) indexed by name. Given a dispatch expression like `e.foo()`, our analysis should compute the set of classes that `e` may evaluate to, search each class's dispatch table for a method named `foo`, and execute it (abstractly) if it exists. Methods are modeled by the `fun` constructor. Methods model the implicit `this` parameter with the first `T` field, the single formal parameter by the second `T` field, and a single return value by the third `T` field. Recall that the function constructor must be contravariant in its domain and covariant in its range, as reflected in the specification.

For this approach to work, our dispatch table abstraction must map between method names and `method` terms, which we do using the Row sort. A Row of base sort `s` (written Row(s)) denotes a partial function from an infinite set of names to terms of sort `s`. Row expressions, which we do not further explain here, are used to model record types with width and depth subtyping.

Figure 5 shows new rules for object initialization and method dispatch. These rules in conjunction with the rules in Figure 4 comprise our receiver class analysis. For a class `C`, rule (New) returns a `class` expression with label ℓ_C. The `dispatch` component of this expression is a row mapping labels ℓ_{m_i} to `methods` for each method $\mathtt{m}_i$ defined in `C`. To remain consistent with the type judgments for Andersen's analysis (where the result of each type judgment is an l-value) we wrap the resulting class in a `ref` constructor. Note that since our analysis is context-insensitive, each instance of `new C` occurring in the program yields the

$$\frac{}{\Gamma \vdash \texttt{new C} : ref(0, class(\ell_C, < \ell_{m_i} : fun(\mathcal{X}_{this}, \mathcal{X}_{arg}, \mathcal{X}_{ret}) \ldots >), \overline{1})} \text{ (New)}$$

$$\frac{\begin{array}{cc} \Gamma \vdash e_1 : \tau_1 & \Gamma \vdash e_2 : \tau_2 \\ \tau_1 \subseteq ref(1, \mathcal{T}_1, \overline{0}) & \tau_2 \subseteq ref(1, \mathcal{T}_2, \overline{0}) \\ \multicolumn{2}{c}{\mathcal{T}_1 \subseteq class(1, < \ell_m : fun(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_{ret}) \ldots >)} \end{array}}{\Gamma \vdash e_1\texttt{.m}(e_2) : ref(0, \mathcal{T}_{ret}, \overline{1})} \text{ (Dispatch)}$$

Fig. 5. Rules for receiver class analysis (add to the rules from Figure 4)

Table 1. Benchmark data for Andersen's analysis

Benchmark	Description	Preproc LOC	Andersen(s)		
			BANE(+gc)	BANE	BANSHEE
gs	Ghostscript	437211	35.5	27.0	6.9
spice	Circuit simulation program	849258	14.0	11.3	3.0
pgsql	PostgreSQL	1322420	44.8	34.9	6.0
gimp	GIMP v1.1.14	7472553	1688.8	962.9	20.2
linux	Linux v2.4 (default config)	3784959	—	—	54.5

same `class` expression, which is created when the definition of `C` is analyzed. In (Dispatch), e_1 is analyzed and assumed to contain a set of `class`es. For each class defining a method `m` (i.e. the associated `dispatch` row contains a mapping for label ℓ_m), the corresponding method body is selected and constrained so actual parameters flow into the formal parameters and the return value of the function (lifted to an l-value) is the result of the entire expression.

We conclude the case study by noting that BANSHEE can be used to explore many analysis issues that we have not illustrated here. For example, field-sensitive analyses can be implemented using BANSHEE's Row sort to model structures. Polymorphic recursive analyses can be implemented using a BANSHEE-based library for context-free language reachability [15]. BANSHEE also has a modular design that allows new sorts to be added to the system in case an analysis demands a customized set of resolution rules.

7 Experiments

To demonstrate the scalability and performance of BANSHEE, we implemented field- and context-insensitive Andersen's analysis for C and tested it on several large benchmarks.[5] We also ran the same analysis implemented with the BANE toolkit. While BANE is written in SML and BANSHEE in C, among other low-level differences, the comparison does demonstrate BANSHEE's engineering

[5] All experiments were run on a 2.80 GHz Intel Xeon machine with 4 GB of memory running Red Hat Linux.

improvements. Table 1 shows wall clock execution times in seconds for the benchmarks. Benchmark size is measured in preprocessed lines of code (the two largest benchmarks, gimp and Linux, are approximately 430,000 and 2.2 million source lines of code, respectively). We compiled the Linux benchmark using a "default" configuration that answers "yes" to all interactive configuration options. All reported times for this experiment include the time to calculate the transitive lower bounds of the constraint graph, which simulates points-to queries on all program variables [9]. Parse times are not included. Interestingly, a significant fraction of the analysis time for the Bane implementation is spent in garbage collection, which may be because almost all of the objects allocated during constraint resolution (nodes and edges in the constraint graph) are live for the entire analysis. We also report (in the column labeled Bane) the wall clock execution time for Andersen's analysis exclusive of garbage collection. The C front-end used in the Bane implementation cannot parse the Linux source, so no number is reported for that benchmark. Although it is difficult to compare to other implementations of Andersen's analysis using wall-clock execution time, we note that our performance appears to be competitive with the fastest hand-optimized Andersen's implementation for answering all points-to queries [12].

We also evaluated the strategy described for backtracking-based incremental analysis (Section 4) by running Andersen's analysis on CQual, a type qualifier inference tool. CQual contains approximately 45,000 source lines of C code (250,000 lines preprocessed). We chose CQual because of our familiarity with its build process: without manual guidance, it is difficult to compile and analyze multiple versions of a code base spanning several years. We looked at each of the 13 commits made to CQual's CVS repository from November 2003 to May 2004 that modified at least one source file and compiled successfully. For each commit we report three different numbers (Figure 6(a))[6]:

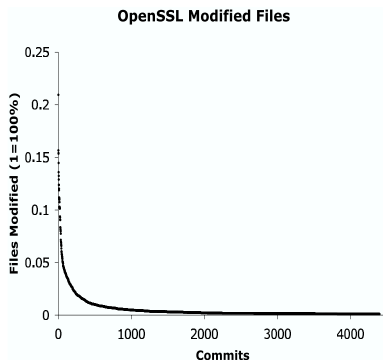
- Column 3: Andersen's analysis run from scratch; this is the analysis time assuming no backtracking is available.
- Column 4: Incremental Andersen's analysis, assuming that analysis of the previous commit is available. To compute this number, we take the previous analysis, backtrack (pop the analysis stack) to the earliest modified file, and re-analyze all files popped off of the stack, placing the modified files on top (as described in Section 4). The initial stack (for the full analysis of the first commit) contained the files in alphabetical order.
- Column 5: Incremental Andersen's analysis, assuming that the files modified during this commit are already on the top of the analysis stack. To compute this number, we pop and reanalyze just the modified files.

Column 4 gives the expected benefit of backtracking if the analysis is run only once per commit. However, if the developer runs the analysis multiple times before committing (each time the code is compiled, or each time the code is edited in an interactive environment) then Column 5 gives a lower bound on the eventual expected benefit. To see this, assume that a single source file is modified

[6] These times do not include the time to serialize or deserialize the constraints.

Commit Date	Files Modified	All Files(s)	Cross Commit	Modified Only(s)
11-16	0/58	2.0	-	-
11-17	1/58	2.0	0.9	0.05
12-03	1/58	2.0	0.9	0.15
12-10	1/58	2.0	1.0	0.04
12-11	1/58	2.3	2.0	0.09
12-12	5/58	2.3	1.8	0.51
2-29	1/58	2.0	0.5	0.08
3-05	1/58	2.0	0.9	0.06
3-05	25/59	2.0	2.0	1.0
3-05	5/60	2.0	2.4	0.27
3-11	4/60	2.4	1.0	0.5
3-22	3/61	2.0	0.14	0.14
5-03	2/61	2.0	1.2	0.13

(a) Data for CQual experiment



(b) OpenSSL files modified per commit

Fig. 6. Backtracking experiments

during an editing task. The first time the analysis is run, that source file may be placed on the bottom of the stack, so after a program edit, a complete reanalysis might be required. Subsequently, however, that file is on top of the stack and we only pay the cost of reanalyzing a single file. In general, if n files are modified in an editing task, at worst we analyze the entire code base n times (to move each file one by one from the bottom to the top of the stack) and subsequently only pay (at most) the cost to reanalyze the n modified files.

Backtracking, then, can be an effective incremental analysis technique as long as only a small fraction of the files in a code base is modified per editing task. Figure 6(a) shows this property holds for CQual. To test this hypothesis on a larger, more active code base with more than a few developers, we looked at the CVS history for OpenSSL, which contains over 4000 commits that modify source code. For each commit, we recorded the percentage of source files modified. Figure 6(b) shows a plot of the sorted data. The percentage of files modified obeys a power law: very infrequently, between 5 and 25 percent of the files are modified, but in the common case, less than .1 percent of the files are modified. We have confirmed similar distributions hold for other code bases as well.

8 Related Work

Many related frameworks have been used to specify static analyses. In [26], modal logic is used as a specification language to compile specialized implementations of dataflow analyses. Datalog [4] is a database query language based on logic programming that has recently received attention as a specification language for static analyses. The subset of pure set constraints implemented in Banshee is equivalent to chain datalog [31] and also context-free language reachability [18]. There are also obvious connections to bottom-up logic [17]. Implementations of these frameworks have been applied to solve static analysis problems. The bddbddb system is a deductive database that uses a binary-decision diagram library

as its back-end [30]. Other toolkits that use BDD back-ends include CrocoPat [3] and Jedd [16]. An efficient algorithm for Dyck context-free language reachability has been shown to be useful for solving various flow analysis problems [21]. A demand-driven version of the algorithm also exists [13], though we have not so far seen a fully incremental algorithm described. Our description of a precise incremental algorithm, as well as our backtracking algorithm, can be applied to Dyck-CFLR problems via a reduction in [15].

We are not aware of previous work on incrementalizing set constraints, though work on incrementalizing transitive closure is abundant and addresses related issues [6,22]. The CLA (compile, link, analyze) [12] approach to analyzing large code bases supports a form of file-granularity incrementality similar to traditional compilers: modified files can be recompiled and linked to any unchanged object files. This approach has some advantages. For example, since CLA doesn't save any analysis results, object file formats are simpler, and there is no need for persistence. However, CLA defers all analysis work until after the link phase, so the only savings is the cost of parsing and producing object files.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.
2. A. Aiken, M. Fähndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proceedings of the Second International Workshop on Types in Compilation (TIC'98)*, 1998.
3. D. Beyer, A. Noack, and C. Lewerentz. Simple and efficient relational querying of software structures. In *Proceedings of the 10th Working Conference on Reverse Engineering*, page 216. IEEE Computer Society, 2003.
4. S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.
5. M. Das. Unification-based pointer analysis with directional assignments. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 2000.
6. C. Demetrescu and G. F. Italiano. Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 381. IEEE Computer Society, 2000.
7. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 109–121, 1986.
8. M. Fähndrich and A. Aiken. Program analysis using mixed term and set constraints. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 114–126, London, United Kingdom, 1997. Springer-Verlag.
9. M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Montreal, Canada, June 1998.
10. Z. Galil and G. F. Italiano. A note on set union with arbitrary deunions. *Information Processing Letters*, 37(6):331–335, 1991.

11. D. Gay and A. Aiken. Language support for regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, 2001.
12. N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.
13. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 104–115. ACM Press, 1995.
14. J. Kodumal. Banshee: A toolkit for constructing constraint-based analyses. http://banshee.sourceforge.net, 2005.
15. J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 207–218. ACM Press, 2004.
16. O. Lhoták and L. Hendren. Jedd: A BDD-based relational extension of Java. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation.* ACM Press, 2004.
17. D. McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.
18. D. Melski and T. Reps. Interconvertbility of set constraints and context-free language reachability. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 74–89. ACM Press, 1997.
19. J. Ragan-Kelley. Personal communication, November 2004.
20. J. Ragan-Kelley. *Practical Interactive Lighting Design for RenderMan Scenes.* Undergraduate thesis, Stanford University, Department of Computer Science, 2004.
21. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, January 1995.
22. L. Roditty. A faster and simpler fully dynamic transitive closure. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 404–412. Society for Industrial and Applied Mathematics, 2003.
23. A. Sayeed. Proshee. http://proshee.sourceforge.net, 2005.
24. M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1997.
25. B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, Jan. 1996.
26. B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21(2):115–139, 1993.
27. G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 183–194, 2005.
28. Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–95. ACM Press, 2000.
29. J. Westbrook and R. E. Tarjan. Amortized analysis of algorithms for set union with backtracking. *SIAM Journal on Computing*, 18(1):1–11, 1989.

30. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM Press, June 2004.
31. M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 230–242. ACM Press, 1990.

A Tracked Reference Implementation

We include a complete listing of the tracked reference datatype in OCaml.

```
module S = Stack
exception Tick

type clock = {
  mutable time : int;
  repository : (unit -> unit) S.t
}
type 'a tref = clock * 'a ref

let tref (clk: clock) (v :'a) : 'a tref =
  (clk, ref v)
let read (clk,r: 'a tref) : 'a =
  !r
let write (clk,r: 'a tref) (v : 'a) : unit =
  let old_v = !r in
  let closure = fun () -> r := old_v in
    begin
      S.push closure clk.repository;
      r := v
    end
let clock () : clock =
  {
    time = 0;
    repository = S.create ()
  }
let time (clk : clock) : int =
  clk.time
let tick (clk : clock) : unit =
  let closure =
    fun () -> raise Tick in
    begin
      S.push closure clk.repository;
      clk.time <- clk.time + 1;
    end
let rollback (clk : clock) : unit =
  try
    while (not (S.is_empty clk.repository)) do
      let closure = (S.pop clk.repository) in
        closure()
    done
  with Tick -> (clk.time <- clk.time - 1)
```

A Generic Framework for Interprocedural Analysis of Numerical Properties

Markus Müller-Olm[1] and Helmut Seidl[2]

[1] Universität Dortmund, Fachbereich Informatik, LS 5,
Baroper Str. 301, 44221 Dortmund, Germany
markus.mueller-olm@cs.uni-dortmund.de
[2] TU München, Institut für Informatik, I2, 80333 München, Germany
seidl@in.tum.de

Abstract. In his seminal paper [5], Granger presents an analysis which infers linear congruence relations between integer variables. For affine programs without guards, his analysis is *complete*, i.e., infers *all* such congruences. No upper complexity bound, though, has been found for Granger's algorithm. Here, we present a variation of this analysis which runs in polynomial time. Moreover, we provide an interprocedural extension of this algorithm. These algorithms are obtained by means of multiple instances of a general framework for constructing interprocedural analyses of numerical properties. Finally, we indicate how the analyses can be enhanced to deal with equality guards interprocedurally.

1 Introduction

In recent years, a growing interest in the design of very precise analyses of numerical properties of programs could be observed. On the one hand, this comes from a revived interest in aggressive program optimizations as demanded by low-cost embedded processors. On the other hand when designing and implementing critical applications, we are faced with a need for certifying absence of certain program errors [2,11] or security vulnerabilities such as buffer-overflows [3,15].

Here, we concentrate on equality-based numerical properties. Such properties are particularly useful, e.g., for induction variable detection or identification of data alignments [1]. This type of analysis has been pioneered by Karr in [9] where he presents a first intraprocedural analysis of valid affine relations over a field. Karr's analysis maintains for every program point a vector space of valid affine relations. Fifteen years later, his analysis was generalized by Granger [4,5]. Since Granger uses $\mathbb{Z}$ instead of $\mathbb{Q}$, his intraprocedural analysis not only returns valid affine relations but also valid affine congruence relations — with the draw-back, perhaps, that no upper complexity bound is known. Granger's analysis also differs from Karr's in that Granger first determines a linear (in fact affine) abstraction of the sets of intraprocedurally reachable states from which the set of valid relations then is derived in a second step. A forward accumulation of the abstracted collecting semantics is also used by Müller-Olm and Seidl in [12] where (in absence of equality guards) the run-time of Karr's analysis algorithm is improved and also the sizes of occurring numbers is bounded. The same authors also provide the first precise interprocedural extension of Karr's analysis [13] and show how

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 235–250, 2005.
© Springer-Verlag Berlin Heidelberg 2005

it can be adapted to work not only over fields but also over modular rings $\mathbb{Z}_m$ where $m = 2^w$ as used by standard programming languages like Java [14]. In [6,7], Gulwani and Necula re-consider Karr's analysis problem. In order to improve on the complexity of the analysis, they propose randomization. In particular, sizes of occurring numbers are bounded by computing modulo random primes.

In this paper, we present general methods how intraprocedural analyses of numerical properties can be constructed which naturally extend to interprocedural analyses of the same properties. Our framework is parametric in the ring within which the computation of the analysis is performed. For the case of affine relations over fields or modular rings $\mathbb{Z}_m$ (m a power of 2), we subsume versions of the intra- and interprocedural analyses from [12,13] and [14], respectively. Beyond these known analyses, we succeed in deriving an interprocedural extension of Granger's analysis [5] that determines not only all valid affine relations but also all valid congruence relations. We also indicate how the analyses can be enhanced to deal with equality guards interprocedurally.

The immediate interprocedural extension of Granger's analysis as provided by the general framework shares with Granger's original algorithm the draw-back of performing fixpoint iterations over complete lattices with unbounded (though finite) ascending chains. In order to improve on this, we propose a new algorithm which, in absence of procedures, runs in polynomial time. The new algorithm is based on a careful inspection of Granger's analysis problem which allows us to divide the analysis into one analysis over the field $\mathbb{Q}$ together with several analyses over carefully chosen modular rings.

The paper is organized as follows. In section 2 we introduce affine programs together with their collecting semantics. In section 3 we introduce, for every ring R, the R-*linear abstraction* and show how it can be used to determine valid R-linear relations and also (in case of $\mathsf{R} = \mathbb{Z}$) valid linear congruence relations. In section 4, we then show for every *principal ideal ring* R that the R-linear abstraction of the collecting semantics can be computed precisely and provide complexity bounds for fields and modular rings $\mathbb{Z}_m$. In section 5, we particularly deal with the case $\mathsf{R} = \mathbb{Z}$ and provide an alternative algorithm which (at least in absence of equality guards) determines all intraprocedurally valid linear congruence relations in polynomial time. In the interprocedural case, the new algorithm is polynomial if the length of intermediately occurring numbers is polynomially bounded. In section 6, we finally extend the proposed approach to take equality guards into account. Finally, section 7 summarizes and gives hints on directions of future research.

2 The General Set-Up

We use similar conventions as in [13] and [14] which we recall here for reasons of selfcontainedness. Thus, programs are modeled by systems of non-deterministic flow graphs that can recursively call each other as in Figure 1. Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ be the set of (global) variables the program operates on. In order to cover the various computational domains of interest, we assume that the variables take values in some commutative ring R with 1 element. In the programs we analyze, we assume the basic statements either to be *affine assignments* of the form $\mathbf{x}_j := t_0 + \sum_{i=1}^{k} t_i\mathbf{x}_i$ (with $t_i \in \mathsf{R}$ for $i = 0, \ldots, k$ and $\mathbf{x}_j \in \mathbf{X}$) or *non-deterministic assignments* of the form $\mathbf{x}_j :=?$

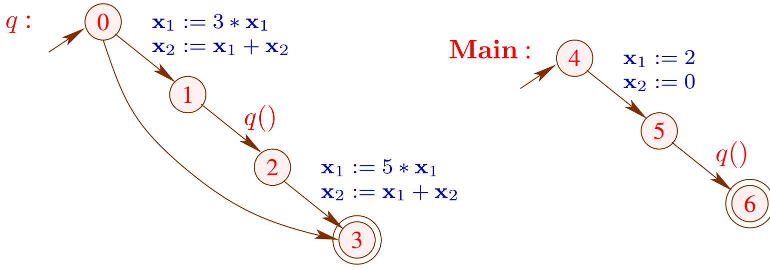


Fig. 1. An interprocedural program

(with $\mathbf{x}_j \in \mathbf{X}$). It is to reduce the number of program points in the example, that we annotated the edges in Figure 1 with sequences of assignments. Also, we use assignments $\mathbf{x}_j := \mathbf{x}_j$ which have no effect onto the program state as skip-statements and omit these in pictures. For the moment, skip-statements are used to abstract guards. Later, we will present methods which treat equality guards more precisely. Non-deterministic assignments $\mathbf{x}_j :=?$ can be used as a safe abstraction of statements in a source program which our analysis cannot handle precisely, for example of assignments $\mathbf{x}_j := t$ with non-affine expressions t or of read statements.

In this setting, an *affine program* comprises a finite set Proc of *procedure names* together with one distinguished procedure $\mathbf{Main}$. Execution starts with a call to $\mathbf{Main}$. Each procedure $q \in \mathsf{Proc}$ is specified by a distinct edge-labeled *control flow graph* with a single start point st_q and a single return point ret_q where each edge is either labeled with an assignment or a call to some procedure.

The basic approach of [13,12,14] which we take up here is to construct a precise abstract interpretation of a constraint system characterizing the concrete program semantics. Similar to [5,12], we find it convenient to start from the *collecting* semantics. For that, we model a *state* attained by program execution when reaching a program point or procedure by a k-dimensional (column) vector[1] $x = [x_1, \ldots, x_k]^{\mathrm{t}} \in \mathsf{R}^k$ of ring elements where x_i is the value assigned to variable $\mathbf{x}_i$. For convenience, we consider *extended* states $[1, x_1, \ldots, x_k]^{\mathrm{t}}$ containing an extra 0-th component 1. Then every assignment $\mathbf{x}_j := t$, $\mathbf{x}_j \in \mathbf{X}$, $t \equiv t_0 + \sum_{i=1}^{k} t_i \mathbf{x}_i$, induces a *linear* transformation $[\![\mathbf{x}_j := t]\!] : \mathsf{R}^{k+1} \to \mathsf{R}^{k+1}$ of the extended state which is described by the matrix:

$$[\![\mathbf{x}_j := t]\!] \;=\; \left[\begin{array}{c|c} I_j & 0 \\ \hline t_0 \ldots t_{j-1} & t_j \ldots t_k \\ \hline 0 & I_{k-j} \end{array}\right]$$

where I_j is the identity matrix in R^{j^2}. This definition is readily extended to sets of extended states. Composition of transformations is captured by matrix multiplication. Since linear mappings are closed under composition, the effect of a single run can be represented by one matrix in $\mathsf{R}^{(k+1)^2}$. Since in general, procedures have multiple runs, we model their semantics by *sets* of linear transformations. These are characterized by the constraint system $\mathcal{E}_{\mathsf{R}}$:

[1] The superscript “t” denotes the *transpose* operation which mirrors a matrix at the main diagonal and changes a row vector into a column vector (and vice versa).

$$
\begin{array}{lll}
[\mathcal{E}_{\mathsf{R}}1] & \mathcal{E}_{\mathsf{R}}(q) \supseteq \mathcal{E}_{\mathsf{R}}(\mathsf{ret}_q) & \\
[\mathcal{E}_{\mathsf{R}}2] & \mathcal{E}_{\mathsf{R}}(\mathsf{st}_q) \supseteq \{I_{k+1}\} & \\
[\mathcal{E}_{\mathsf{R}}3] & \mathcal{E}_{\mathsf{R}}(v) \supseteq \mathcal{E}_{\mathsf{R}}(u) \cdot \{[\![\mathbf{x}_j := t]\!]\} & \text{if edge } (u,v) \text{ is labeled } \mathbf{x}_j := t \\
[\mathcal{E}_{\mathsf{R}}4] & \mathcal{E}_{\mathsf{R}}(v) \supseteq \mathcal{E}_{\mathsf{R}}(u) \cdot \{[\![\mathbf{x}_j := c]\!] \mid c \in \mathsf{R}\} & \text{if edge } (u,v) \text{ is labeled } \mathbf{x}_j :=? \\
[\mathcal{E}_{\mathsf{R}}5] & \mathcal{E}_{\mathsf{R}}(v) \supseteq \mathcal{E}_{\mathsf{R}}(u) \cdot \mathcal{E}_{\mathsf{R}}(q) & \text{if edge } (u,v) \text{ calls } q
\end{array}
$$

The variable $\mathcal{E}_{\mathsf{R}}(q)$ is meant to capture the set of effects of the procedure q. By the constraints $\mathcal{E}_{\mathsf{R}}1$, this value is obtained as the set of transformations $\mathcal{E}_{\mathsf{R}}(\mathsf{ret}_q)$ for the return point ret_q of q. According to $\mathcal{E}_{\mathsf{R}}2$, this accumulation starts at the start point st_q with the identity transformation. The constraints $\mathcal{E}_{\mathsf{R}}3$ and $\mathcal{E}_{\mathsf{R}}4$ deal with affine and nondeterministic assignments, respectively, while the constraints $\mathcal{E}_{\mathsf{R}}5$ correspond to calls.

Given the effects of procedures, we characterize the sets of extended states reaching program points and procedures by the constraint system $\mathcal{C}_{\mathsf{R}}$:

$$
\begin{array}{lll}
[\mathcal{C}_{\mathsf{R}}1] & \mathcal{C}_{\mathsf{R}}(\mathbf{Main}) \supseteq \{1\} \times \mathsf{R}^k & \\
[\mathcal{C}_{\mathsf{R}}2] & \mathcal{C}_{\mathsf{R}}(q) \supseteq \mathcal{C}_{\mathsf{R}}(u) & \text{if edge } (u,_) \text{ calls } q \\
[\mathcal{C}_{\mathsf{R}}3] & \mathcal{C}_{\mathsf{R}}(\mathsf{st}_q) \supseteq \mathcal{C}_{\mathsf{R}}(q) & \\
[\mathcal{C}_{\mathsf{R}}4] & \mathcal{C}_{\mathsf{R}}(v) \supseteq [\![\mathbf{x}_j := t]\!](\mathcal{C}_{\mathsf{R}}(u)) & \text{if edge } (u,v) \text{ is labeled } \mathbf{x}_j := t \\
[\mathcal{C}_{\mathsf{R}}5] & \mathcal{C}_{\mathsf{R}}(v) \supseteq \bigcup\{[\![\mathbf{x}_j := c]\!](\mathcal{C}_{\mathsf{R}}(u)) \mid c \in \mathsf{R}\} & \text{if edge } (u,v) \text{ is labeled } \mathbf{x}_j :=? \\
[\mathcal{C}_{\mathsf{R}}6] & \mathcal{C}_{\mathsf{R}}(v) \supseteq \mathcal{E}_{\mathsf{R}}(q)(\mathcal{C}_{\mathsf{R}}(u)) & \text{if edge } (u,v) \text{ calls } q
\end{array}
$$

The constraint $\mathcal{C}_{\mathsf{R}}1$ indicates that we start before the call of $\mathbf{Main}$ with the full (extended) state space. The constraints $\mathcal{C}_{\mathsf{R}}2$ indicate that the extended states reaching a procedure includes all extended states reaching its calls and the constraints $\mathcal{C}_{\mathsf{R}}3$ state that the extended states reaching a call to a procedure also reach its start point. The constraints $\mathcal{C}_{\mathsf{R}}4$ through $\mathcal{C}_{\mathsf{R}}6$ then are completely analogous to a usual forward propagating definition of the *intra-procedural* collecting semantics only that at a call edge the set of transformations obtained for the called procedure is applied (constraints $\mathcal{C}_{\mathsf{R}}6$).

By the fixpoint theorem of Knaster-Tarski, the constraint systems $\mathcal{E}_{\mathsf{R}}$ and $\mathcal{C}_{\mathsf{R}}$ have least solutions. For convenience, we denote the components of these least solutions by $\mathcal{E}_{\mathsf{R}}(X)$, and $\mathcal{C}_{\mathsf{R}}(X)$, respectively ($X$ a procedure name or program point).

3 The Linear Abstraction

Program analyses of numerical program properties are based on abstractions of subsets of vectors. Here, we consider the abstraction of a set $V \subseteq \mathsf{R}^{k+1}$ of extended states by the R-*linear closure* of V:

$$\alpha_{\mathsf{R}}(V) = \langle V \rangle_{\mathsf{R}} = \{\lambda_1 v_1 + \ldots + \lambda_s v_s \mid s \geq 0, \lambda_i \in \mathsf{R}, v_i \in V\}\,.$$

Due to the extension of states by an extra 0-th component, the abstraction adds all *linear* combinations of vectors in V – with the understanding that only those vectors in the closure are meaningful whose 0-th components equal 1. We remark that $\alpha_{\mathsf{R}}(V)$ is closed under vector addition and multiplication with ring elements $r \in \mathsf{R}$. Such sets are called R-*modules* where the set $\langle V \rangle_{\mathsf{R}}$ is the R-module *generated* by V. It is well-known that for any r, the R-submodules of R^r are closed under intersection. Ordered by set inclusion

(which we denote by $\sqsubseteq$ in the context of submodules) they thus form a complete lattice $\mathbf{Sub}(\mathsf{R}^r)$, like the linear subspaces of $\mathbb{F}^r$ for a field $\mathbb{F}$. The least element of $\mathbf{Sub}(\mathsf{R}^r)$ is $\{\mathbf{0}\}$ consisting of the zero vector only, the greatest element is R^r itself. The least upper bound of two R-submodules M_1, M_2 is

$$M_1 \sqcup M_2 = \langle M_1 \cup M_2 \rangle_{\mathsf{R}} = \{m_1 + m_2 \mid m_i \in M_i\} .$$

The linear abstraction has been extensively studied for different rings. In [5], it is used with $\mathsf{R} = \mathbb{Z}$ to analyze linear congruence relations. In [12], this abstraction is applied for fields to speed up Karr's analysis [9] of affine relations. Interestingly, the interprocedural analyses of affine relations [13,14] over fields and modular rings $\mathbb{Z}_m$, $m = 2^w$, do not directly rely on abstractions of the collecting semantics but on linear abstractions of sets of weakest precondition transformers.

In general, we are interested in numerical properties P which invariantly hold for all (extended) states x in the collecting semantics at a given program point. Clearly, the linear abstraction can only be used to detect properties which are invariant under linear combinations of the extended state or, equivalently, affine combinations of the program state. In particular, this is the case for *affine* relations between program variables like, e.g., $2 - 4\mathbf{x}_1 + 3\mathbf{x}_2 = 0$. Since we work with extended states, we can rely on the simpler *linear* relations on extended states here. In general, a linear relation over a ring R is a (row) vector $a = [a_0, \ldots, a_k]$ where $x = [x_0, \ldots, x_k]^{\mathrm{t}}$ satisfies a iff $a \cdot x = \sum_{i=0}^{k} a_i x_i = 0$. The set of affine relations satisfied by a set of states coincides with the set of linear relations satisfied by the corresponding set of extended states. We observe:

Fact 1. *For every ring* R *the following holds:*

1. *For every row vector a, the set $\{x \in \mathsf{R}^{k+1} \mid a \cdot x = 0\}$ is an* R*-module.*
2. *For every set $G \subseteq \mathsf{R}^{k+1}$,*

$$\langle G \rangle_{\mathsf{R}}^{\perp} =_{\mathsf{def}} \{a \mid \forall x \in G : a \cdot x = 0\} = \{a \mid \forall x \in \langle G \rangle_{\mathsf{R}} : a \cdot x = 0\} .$$

Moreover, the set $\langle G \rangle_{\mathsf{R}}^{\perp}$ is an R*-module.* □

Assume that the R-module $\langle \mathcal{C}_{\mathsf{R}}(X) \rangle_{\mathsf{R}}$ is generated by the finite set $G \subseteq \mathsf{R}^{k+1}$. Then by fact 1, we can determine the set of all valid linear relations at X as the set of all solutions of the homogeneous system of equations:

$$\mathbf{a} \cdot x = 0 \,, \quad x \in G$$

where $\mathbf{a} = [\mathbf{a}_0, \ldots, \mathbf{a}_k]$ is a row vector of variables. Here, we are mostly interested in *principal ideal rings* (or PIRs). A principal ring R is a commutative ring with 1 in which every ideal is *principal*. Recall that an *ideal* $I \subseteq \mathsf{R}$ is a subset of R which is closed under addition and multiplication with arbitrary ring elements, i.e., $a + b \in I$ whenever $a, b \in I$ and $r \cdot a \in I$ whenever $a \in I$ and $r \in \mathsf{R}$. An ideal I is principal if it is generated by a single element, i.e., $I = \{r \cdot d \mid r \in \mathsf{R}\}$ for some $d \in \mathsf{R}$. PIRs comprise not only fields but also the integral domain $\mathbb{Z}$ as well as all modular rings $\mathbb{Z}_m$, $m \geq 2$. In [8,16], efficient methods are developed for computing various normal forms of matrices over PIRs. The most notable property of PIRs is that they allow us to solve linear systems of equations by a generalized Gaussian elimination algorithm. Of particular importance is the integral domain $\mathbb{Z}$. Assume $G \subseteq \mathbb{Z}^{k+1}$ is a set of integer vectors. Then the set of all

linear relations which are valid for G is (up to multiplication with constants) identical to the set of linear relations which are valid over the $\mathbb{Q}$-module generated by G:

Fact 2. *For every subset $G \subseteq \mathbb{Z}^{k+1}$ of column vectors and every row vector $a \in \mathbb{Z}^{k+1}$, the following statements are equivalent:*

1. $a \cdot x = 0$ *for all* $x \in G$;
2. $a \cdot x = 0$ *for all* $x \in \langle G \rangle_{\mathbb{Z}}$;
3. $a \cdot x = 0$ *for all* $x \in \langle G \rangle_{\mathbb{Q}}$. □

Assume we want to determine the set of valid $\mathbb{Z}$-linear relations at a program point X. By fact 2, it suffices to determine the linear relations which are valid for $\langle \mathcal{C}_{\mathbb{Z}}(X) \rangle_{\mathbb{Q}}$. Since $\mathbb{Q}$ is a field, these can be computed efficiently with the techniques from [13,12]. It therefore does not pay off to determine the (complicated) $\mathbb{Z}$-linear closure of the collecting semantics if we are interested in linear relations only.

In [5], however, Granger considers a more general form of properties, namely, *linear congruence relations*. A linear congruence equation is an equation $a \cdot \mathbf{x} \equiv 0\ [m]$ where $a \in \mathbb{Z}$ is a row vector and $m > 0$ is the integer modulus. The column vector $x \in \mathbb{Z}^{k+1}$ satisfies the congruence relation iff $a \cdot x \equiv 0\ [m]$ or, equivalently, $a \cdot x + mz = 0$ for some $z \in \mathbb{Z}$. A linear relation of the extended state can be seen as a particular linear congruence relation if we allow m to equal 0. If $m > 1$, we can assume that all components of a are in the range $\{0, \ldots, m-1\}$. The set of all x satisfying a linear congruence relation is closed under addition and multiplication with elements of $\mathbb{Z}$ and therefore a $\mathbb{Z}$-module. In [5], Granger shows that every $\mathbb{Z}$-module can also be represented as the set of solutions of a finite number of linear congruence relations. For later use, we provide a refinement of his characterization. We introduce the following auxiliary notions. Assume that $G \subseteq \mathbb{Z}^r$ is a set of q linearly independent[2] column vectors. Let $V \subseteq \mathbb{Z}^{r \cdot q}$ denote the matrix formed by the vectors in G. Using generalized Gaussian elimination, some unimodular matrix[3] $T \in \mathbb{Z}^{r^2}$ can be constructed such that $T \cdot V = \left[\frac{D}{0}\right]$ for an upper triangular square matrix D. Then we define $\mathsf{det}(G)$ as the absolute value of the determinant of D. It follows from uniqueness of the Hermite normal form [17,16] that this definition is independent of the choice of T. We obtain:

Theorem 1. *Assume $G \subseteq \mathbb{Z}^r$ is a set of linearly independent vectors where $\mathsf{det}(G)$ divides $m > 0$. Let E_0 and E_m denote finite sets of generators for $\langle G \rangle_{\mathbb{Z}}^{\perp}$ and $\langle G \rangle_{\mathbb{Z}_m}^{\perp}$, respectively. Then the following holds:*

1. $\langle G \rangle_{\mathbb{Z}}$ *is the set of solutions of the system*

$$a \cdot \mathbf{x} = 0, a \in E_0, \qquad b \cdot \mathbf{x} \equiv 0\ [m], b \in E_m$$

2. *Another linear congruence relation $b' \cdot \mathbf{x} \equiv 0\ [m']$ is satisfied by all vectors in G iff the following holds. If $m' = 0$ then $b' \in \langle E_0 \rangle_{\mathbb{Z}}$. Otherwise, let h denote the least common multiple of m and m' where $m \cdot d = h$ and $m' \cdot d' = h$. Then $d' \cdot b'$ is contained in $\langle E_0 \cup \{d \cdot b \mid b \in E_m\} \rangle_{\mathbb{Z}_h}$.*

[2] Recall that G is linearly independent over $\mathbb{Q}$ iff G is linearly independent over $\mathbb{Z}$.

[3] An integer matrix is *unimodular* iff its determinant equals ± 1.

For a proof of this theorem, see appendix A. By the second statement, the sets E_0 and E_m allow us, for every other modulus m', to determine a finite set E' of generators of all valid linear relations modulo m'. First, we construct the set $E = E_0 \cup \{d \cdot b \mid b \in E_m\}$ where $h = d \cdot m$ is the least common multiple of m and m'. The idea is now to determine E' as a finite set of generators of all $\mathbb{Z}_h$-linear combinations of vectors in E which contain $d' = \frac{h}{m'}$ as a factor. For this, let V denote the matrix whose rows are formed by the vectors in E. Then a vector v is a linear combination of the vectors in E which contains d' as a factor iff $v = y \cdot V$ for some $y \in \mathbb{Z}_h^{|E|}$ such that $m' \cdot (y \cdot V) \equiv \mathbf{0}\ [h]$. Thus, we first compute generators $b_1, \ldots, b_q \in \mathbb{Z}_h^{|E|}$ for the module of solutions of the equation system $\mathbf{y}\,(m' \cdot V) \equiv \mathbf{0}\ [h]$. The vectors $b_i\, V$ can be written as $b_i\, V = d' b'_i$ — giving us the set $E' = \{b'_1, \ldots, b'_q\}$ of generators for all valid linear relations modulo m'.

Theorem 1 allows us to compute the linear congruence relations which are valid at X from the $\mathbb{Z}$-linear closure of $\mathcal{C}_{\mathbb{Z}}(X)$, the set of extended states reaching X. Our new observation is that, instead of computing the $\mathbb{Z}$-linear closure of the reachable states, we can decompose the analysis into an analysis returning all valid linear relations plus an analysis returning all valid linear relations modulo a carefully chosen m. If on the other hand, we are interested in the linear closure of the reachable extended states at X, then we can recover these from the linear equations together with the valid linear equations modulo m by solving an appropriate homogeneous system of equations.

4 Constructing Interprocedural Analyses

We have seen that for affine programs, the effects of procedures are given by sets of linear transformations, or matrices. Matrices in turn can be viewed as vectors — only with quadratically many components. We therefore can use the same abstraction α_{R} for effects which we use for sets of extended state vectors. By applying α_{R} to the constraint systems $\mathcal{E}_{\mathsf{R}}$ and $\mathcal{C}_{\mathsf{R}}$, we obtain constraint systems $\mathcal{E}^\sharp_{\mathsf{R}}$ and $\mathcal{C}^\sharp_{\mathsf{R}}$:

$[\mathcal{E}^\sharp_{\mathsf{R}}1]$	$\mathcal{E}^\sharp_{\mathsf{R}}(q) \sqsupseteq \mathcal{E}^\sharp_{\mathsf{R}}(\mathsf{ret}_q)$	
$[\mathcal{E}^\sharp_{\mathsf{R}}2]$	$\mathcal{E}^\sharp_{\mathsf{R}}(\mathsf{st}_q) \sqsupseteq \langle\{I_{k+1}\}\rangle_{\mathsf{R}}$	
$[\mathcal{E}^\sharp_{\mathsf{R}}3]$	$\mathcal{E}^\sharp_{\mathsf{R}}(v) \sqsupseteq \mathcal{E}^\sharp_{\mathsf{R}}(u) \cdot \langle\{[\![\mathbf{x}_j := t]\!]\}\rangle_{\mathsf{R}}$	if edge (u, v) is labeled $\mathbf{x}_j := t$
$[\mathcal{E}^\sharp_{\mathsf{R}}4]$	$\mathcal{E}^\sharp_{\mathsf{R}}(v) \sqsupseteq \mathcal{E}^\sharp_{\mathsf{R}}(u) \cdot \langle\{[\![\mathbf{x}_j := 0]\!], [\![\mathbf{x}_j := 1]\!]\}\rangle_{\mathsf{R}}$	if edge (u, v) is labeled $\mathbf{x}_j :=?$
$[\mathcal{E}^\sharp_{\mathsf{R}}5]$	$\mathcal{E}^\sharp_{\mathsf{R}}(v) \sqsupseteq \mathcal{E}^\sharp_{\mathsf{R}}(u) \cdot \mathcal{E}^\sharp_{\mathsf{R}}(q)$	if edge (u, v) calls q

As in [13,12], the abstract effect of a non-deterministic assignment $\mathbf{x}_j :=?$ can be modeled by the span of the two transformations $[\![\mathbf{x}_j := 0]\!]$ and $[\![\mathbf{x}_j := 1]\!]$.

The constraint system $\mathcal{E}^\sharp_{\mathsf{R}}$ closely resembles the corresponding constraint systems as presented in [13] and [14]. There, however, the accumulated transformations are interpreted as *weakest precondition transformers* and therefore accumulated from the rear. The constraint system now accumulates values in a forward fashion. Accordingly, the second constraint system $\mathcal{C}^\sharp_{\mathsf{R}}$ is in the spirit of the forward *intraprocedural* accumulation as used, e.g., in [12]. Thus, in contrast to [13,14], the second constraint system *directly* speaks about abstract sets of values and not about abstract sets of transformations:

$$
\begin{array}{llll}
[\mathcal{C}_{\mathsf{R}}^{\sharp}1] & \mathcal{C}_{\mathsf{R}}^{\sharp}(\mathbf{Main}) & \sqsupseteq \mathsf{R}^{k+1} & \\
[\mathcal{C}_{\mathsf{R}}^{\sharp}2] & \mathcal{C}_{\mathsf{R}}^{\sharp}(q) & \sqsupseteq \mathcal{C}_{\mathsf{R}}^{\sharp}(u) & \text{if edge } (u, _) \text{ calls } q \\
[\mathcal{C}_{\mathsf{R}}^{\sharp}3] & \mathcal{C}_{\mathsf{R}}^{\sharp}(\mathsf{st}_q) & \sqsupseteq \mathcal{C}_{\mathsf{R}}^{\sharp}(q) & \\
[\mathcal{C}_{\mathsf{R}}^{\sharp}4] & \mathcal{C}_{\mathsf{R}}^{\sharp}(v) & \sqsupseteq [\![\mathbf{x}_j := t]\!]\,(\mathcal{C}_{\mathsf{R}}^{\sharp}(u)) & \text{if edge } (u, v) \text{ is labeled } \mathbf{x}_j := t \\
[\mathcal{C}_{\mathsf{R}}^{\sharp}5] & \mathcal{C}_{\mathsf{R}}^{\sharp}(v) & \sqsupseteq [\![\mathbf{x}_j := 0]\!]\,(\mathcal{C}_{\mathsf{R}}^{\sharp}(u)) \sqcup & \\
 & & \quad [\![\mathbf{x}_j := 1]\!]\,(\mathcal{C}_{\mathsf{R}}^{\sharp}(u)) & \text{if edge } (u, v) \text{ is labeled } \mathbf{x}_j :=? \\
[\mathcal{C}_{\mathsf{R}}^{\sharp}6] & \mathcal{C}_{\mathsf{R}}^{\sharp}(v) & \sqsupseteq \mathcal{E}_{\mathsf{R}}^{\sharp}(q)(\mathcal{C}_{\mathsf{R}}^{\sharp}(u)) & \text{if edge } (u, v) \text{ calls } q
\end{array}
$$

By the fixpoint theorem of Knaster-Tarski, the constraint systems $\mathcal{E}_{\mathsf{R}}^{\sharp}$ and $\mathcal{C}_{\mathsf{R}}^{\sharp}$ have least solutions. Again, we denote the components of these least solutions by $\mathcal{E}_{\mathsf{R}}^{\sharp}(X)$ and $\mathcal{C}_{\mathsf{R}}^{\sharp}(X)$, respectively ($X$ a procedure or program point). Abstracting the collecting semantics according to constraint system $\mathcal{C}_{\mathsf{R}}^{\sharp}$ has the advantage that it relies on matrices only for procedure calls. This means that we can take advantage from any improvements on the abstractions, e.g., for guards $g = 0$ (g an affine combination) or non-affine assignments which have been proposed for the intraprocedural analysis [9,5].

Furthermore, we verify that the abstraction commutes with the application and with the composition of transformations. By linearity we have:

Proposition 1. *Let* R *denote a commutative ring with 1. Then:*

1. $\langle\{A\,x \mid x \in V, A \in M\}\rangle_{\mathsf{R}} = \langle\{A\,x \mid x \in \langle V\rangle_{\mathsf{R}}, A \in \langle M\rangle_{\mathsf{R}}\}\rangle_{\mathsf{R}}$
2. $\langle\{A_1\,A_2 \mid A_i \in M_i\}\rangle_{\mathsf{R}} = \langle\{A_1\,A_2 \mid A_i \in \langle M_i\rangle_{\mathsf{R}}\}\rangle_{\mathsf{R}}$

for every set of vectors $V \subseteq \mathsf{R}^{k+1}$ *and sets of matrices* $M, M_1, M_2 \subseteq \mathsf{R}^{(k+1)^2}$. □

By the fixpoint transfer lemma, we therefore obtain from proposition 1, for the constraint systems $\mathcal{E}_{\mathsf{R}}^{\sharp}$ and $\mathcal{C}_{\mathsf{R}}^{\sharp}$:

Theorem 2. *For a program interpreted over a ring* R*, the following holds:*

1. $\mathcal{E}_{\mathsf{R}}^{\sharp}(q) = \langle\mathcal{E}_{\mathsf{R}}(q)\rangle_{\mathsf{R}}$ *for every procedure* q*;*
2. $\mathcal{C}_{\mathsf{R}}^{\sharp}(X) = \langle\mathcal{C}_{\mathsf{R}}(X)\rangle_{\mathsf{R}}$ *for every procedure or program point* X*.* □

Theorem 2 gives a precise characterization of the linear closure of the collecting semantics through a constraint system. Note that for *principal ideal rings* R, the lattice of R-submodules of R^r satisfies the ascending chain condition. If R is a field, the length of every strictly increasing sequence of R-submodules of R^r is bounded by r for dimensional reasons. If R is a modular ring $\mathbb{Z}_m$, then the length of every strictly increasing sequence of R-submodules of R^r can be shown to be bounded by $r \cdot \log(m)$. If R is the ring of integers, the lengths of strictly increasing sequences of R-submodules, though finite, cannot be bounded.

Secondly, we note that every R-submodule M of R^r can be represented by $M = \langle G\rangle_{\mathsf{R}}$ for a set G of at most r generators. Accordingly, inclusion of R-submodules can be reduced to deciding for a vector $v \in \mathsf{R}^r$ whether or not $v \in \langle G\rangle_{\mathsf{R}}$ for a finite subset $G \subseteq \mathsf{R}^r$. If $G = \{v_1, \ldots, v_s\}$, the latter problem consists in deciding whether there exist $\lambda_1, \ldots, \lambda_s \in \mathsf{R}$ such that

$$\lambda_1 v_1 + \ldots + \lambda_s v_s = v$$

Thus, the problem reduces to solving inhomogeneous systems of linear equations. If R is a field, this can be achieved, e.g., by standard Gaussian elimination. Instead, we may rely on reduction to *echelon form* as discussed in [8,16]. Therefore, theorem 2 gives rise to an effective analysis over any *effective* PIR R, i.e., every PIR R where 0 and 1, equality as well as the arithmetic operations and the basic principal ideal operations are computable. The ideal operations we need are a *generalized gcd* and effective methods for solving *one variable* equations $a \cdot \mathbf{x}_1 = b$ with $a, b \in \mathsf{R}$ (see again [8,16] for details). Summarizing, we have:

Theorem 3. *Assume p is an affine program over an effective PIR R. Then the least solutions of the constraint systems $\mathcal{E}^\sharp_\mathsf{R}$ and $\mathcal{C}^\sharp_\mathsf{R}$ are effectively computable.* □

In particular, we obtain interprocedural algorithms for computing the linear closures of the collecting semantics for fields as well as for all modular rings — thus giving us algorithms for computing all valid linear relations. The corresponding run-time complexities for a program of size n with k variables are summarized in figure 2. For simplicity, we have assumed unit cost for every arithmetic operation as well as for the principal ideal operations. The first line reports the results obtained in [12,13], while the result on $\mathbb{Z}_m$

R	intraprocedural	interprocedural
field	$\mathcal{O}(n \cdot k^3)$	$\mathcal{O}(n \cdot k^8)$
$\mathbb{Z}_m$	$\mathcal{O}(n \cdot k^3 \cdot \log(m))$	$\mathcal{O}(n \cdot k^8 \cdot \log(m))$

Fig. 2. Unit cost complexity of computing the R-linear closure

is the generalization of [14] to arbitrary modular rings. Theorem 3 also provides us with an interprocedural generalization of Granger's analysis. The complexity, however, remains unclear here, since ascending chains of $\mathbb{Z}$-modules can have arbitrary lengths.

5 Efficient Linear Congruence Analysis

In this section, we refine the general approach for PIRs for the case $\mathsf{R} = \mathbb{Z}$ in order to obtain a *polynomial time* algorithm for computing all intraprocedurally valid linear congruence relations. This algorithm also extends to a fast interprocedural algorithm — provided that mild restrictions on occurring numbers are satisfied.

Theorem 4. *Assume p is an affine program over $\mathbb{Z}$ of size n with k variables.*

1. *For every program point or procedure X, we can compute a (finite) representation of the set of all linear congruence relations valid at X.*
2. *Intraprocedurally, these representations can be computed in polynomial time.*
3. *Interprocedurally, these representations can be computed in exponential time.*

Proof. Assume the program p has k program variables. The algorithm achieving the explicit complexity bounds is based on theorem 1. It proceeds in three phases.

Phase 1: We compute the least solutions of the constraint systems $\mathcal{E}^\sharp_\mathbb{Q}$ and $\mathcal{C}^\sharp_\mathbb{Q}$. More precisely, we compute for every program point or procedure X, linearly independent subsets $G_\mathcal{E}(X) \subseteq \mathcal{E}_\mathbb{Z}(X)$, $G_\mathcal{C}(X) \subseteq \mathcal{C}_\mathbb{Z}(X)$ such that

$$\mathcal{E}^\sharp_\mathbb{Q}(X) = \langle G_\mathcal{E}(X)\rangle_\mathbb{Q} \qquad \mathcal{C}^\sharp_\mathbb{Q}(X) = \langle G_\mathcal{C}(X)\rangle_\mathbb{Q}$$

Then we determine for every X, a set of generators for the set of all $\mathbb{Z}$-linear relations which are valid at X.

Phase 2: For every X, we determine $m(X)$ as the determinant $\mathsf{det}(G_\mathcal{C}(X))$.

Phase 3: For every X, we solve the constraint systems $\mathcal{E}^\sharp_{\mathbb{Z}_m}$ and $\mathcal{C}^\sharp_{\mathbb{Z}_m}$ for $m = m(X)$. This allows us to determine the $\mathbb{Z}_m$-module $\mathcal{C}^\sharp_{\mathbb{Z}_m}(X)$ and compute a set of generators of the $\mathbb{Z}_m$-linear relations which are valid at X.

We successively discuss the three phases of the algorithm. The first phase is readily implemented by a variant of the algorithm proposed in [13] for solving constraint system $\mathcal{E}^\sharp_\mathbb{Q}$ and (an adapted version of) [12] for then solving $\mathcal{C}^\sharp_\mathbb{Q}$. These algorithms are based on semi-naive fixpoint iteration and generate for every program point or procedure X a basis consisting of matrices from $\mathcal{E}_\mathbb{Z}(X)$ and (extended) states from $\mathcal{C}_\mathbb{Z}(X)$, respectively.

Example 1. Consider, e.g., the program from section 2. We find the matrices:

$$Q_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad Q_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 15 & 0 \\ 0 & 18 & 1 \end{bmatrix} \quad Q_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 225 & 0 \\ 0 & 282 & 1 \end{bmatrix}$$

which are contained in $\mathcal{E}_\mathbb{Z}(q)$ and together generate the vector space $\mathcal{E}^\sharp_\mathbb{Q}(q)$. Using these matrices, we determine a set of generators for the vector-space $\mathcal{C}^\sharp_\mathbb{Q}(6)$ as:

$$z_0 = [1, 2, 0]^t \quad z_1 = [1, 30, 36]^t \quad z_2 = [1, 450, 564]^t$$

□

Since $\langle \mathcal{C}_\mathbb{Z}(X)\rangle_\mathbb{Q} = \langle \mathcal{C}_\mathbb{Q}(X)\rangle_\mathbb{Q}$, fact 1 implies that the $\mathbb{Z}$-module $\langle G_\mathcal{C}(X)\rangle^\perp_\mathbb{Z}$ already equals the $\mathbb{Z}$-module $\langle \mathcal{C}_\mathbb{Z}(X)\rangle^\perp_\mathbb{Z}$, i.e., the set of valid linear equalities.

Let $\Delta_\mathcal{E}, \Delta_\mathcal{C}$ denote the maximal absolute sizes of the entries of the matrices and vectors, respectively, in the sets of generators used by the fixpoint computation over $\mathbb{Q}$. By inspecting the algorithms in [13,12], we find:

$$\Delta_\mathcal{E} \leq 2^{2^{\mathcal{O}(n\cdot k^2)}} \qquad \Delta_\mathcal{C} \leq \Delta_\mathcal{E}^{\mathcal{O}(n\cdot k)}$$

In general, solving the constraint systems $\mathcal{E}^\sharp_\mathbb{Q}$ and $\mathcal{C}^\sharp_\mathbb{Q}$ over $\mathbb{Q}$ thus can be performed by $\mathcal{O}(n \cdot k^8)$ operations using arithmetic for numbers bounded in length by $\mathcal{O}(n \cdot k^2 \cdot \log(\Delta_\mathcal{E}))$. In case of an intra-procedural analysis, we can completely abandon the constraint system $\mathcal{E}^\sharp_\mathbb{Q}$. Adapting the algorithm from [12], we need just $\mathcal{O}(n \cdot k^3)$ arithmetic operations on numbers of length $\mathcal{O}(n \cdot k^2)$.

We turn to phase 2. Given a linearly independent set $G_\mathcal{C}(X)$ of cardinality q, we compute the determinant $m(X) = \mathsf{det}(G_\mathcal{C}(X))$ with a polynomial number of bit operations, e.g., using the methods of Storjohann [17,16]. In our application the length of the computed determinant (and thus also of $\log(m(X))$) is bounded by $\mathcal{O}(n \cdot k^2 \cdot \log(\Delta_\mathcal{E}))$. Let G denote a linearly independent set of generators of $\mathcal{C}^\sharp_\mathbb{Z}(X) = \langle \mathcal{C}_\mathbb{Z}(X)\rangle_\mathbb{Z}$. Since $\langle G\rangle_\mathbb{Q} = \langle G_\mathcal{C}(X)\rangle_\mathbb{Q}$, G has cardinality q as well.

Claim: $\mathsf{det}(G)$ divides $\mathsf{det}(G_\mathcal{C}(X))$.

This central claim together with theorem 1 implies that the set of all linear congruence relations valid at X can be derived from the set of all linear relations valid at X (as computed in phase 1) together with all linear congruence relations modulo $m(X)$ (as computed in phase 3).

We turn to the proof of the claim. Let $V \in \mathbb{Z}^{(k+1)\cdot q}$ and $V' \in \mathbb{Z}^{(k+1)\cdot q}$ denote the coefficient matrices formed by the vectors of $G_{\mathcal{C}}(X)$ and G, respectively. By definition, there are square unimodular matrices $T, T' \in \mathbb{Z}^{(k+1)^2}$ such that $T \cdot V = \left[\frac{D}{0}\right]$ and $T' \cdot V' = \left[\frac{D'}{0}\right]$ for square upper triangular matrices D, D' where the product of the diagonal elements of D and D' equals $\mathsf{det}(G_{\mathcal{C}}(X))$ and $\mathsf{det}(G)$, respectively. Since $G_{\mathcal{C}}(X) \subseteq \langle G\rangle_{\mathbb{Z}}$, there is also a square matrix $S \in \mathbb{Z}^{q^2}$ such that $V = V' \cdot S$. Therefore, $D = T_1 \cdot D' \cdot S$ where T_1 is the left upper $(q \times q)$-submatrix of $T \cdot (T')^{-1}$ and, thus, $\mathsf{det}(D) = \mathsf{det}(T_1) \cdot \mathsf{det}(D') \cdot \mathsf{det}(S)$. Since T_1 and S are integer matrices the claim follows.

Example 2. Starting from the vectors z_0, z_1, z_2 for program point 6 of example 1, we may apply elementary row transformations (over $\mathbb{Z}$) each with determinant 1 to the coefficient matrix of the z_i. Thus, we obtain the matrix:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 4 & 700 \\ 0 & 0 & -84 \end{bmatrix}$$

Thus, the determinant equals $m(6) = 1 \cdot 4 \cdot 84 = 336$ — serving as the modulus for the third stage. Since the three vectors z_0, z_1, z_2 are linearly independent, they span the complete vector space $\mathbb{Q}^3$. Therefore, no non-trivial linear relation holds for every reachable state at program point 6. □

In phase 3, it remains to determine the set of all linear relations *modulo* $m(X)$ which hold for all vectors in $\mathcal{C}^\sharp_{\mathbb{Z}}(X)$. Since taking integers modulo $m(X)$ is a homomorphism, we conclude that the $\mathbb{Z}_{m(X)}$-module $\langle \mathcal{C}^\sharp_{\mathbb{Z}_{m(X)}}(X)\rangle^{\perp}_{\mathbb{Z}_{m(X)}}$ equals the set of all linear congruence relations which are valid at X modulo $m(X)$. Note further that the third phase of fixpoint iteration for the constraint systems over $\mathbb{Z}_{m(X)}$ need not start from scratch but can use the generators computed in the first phase modulo $m(X)$ as start value.

Example 3. We turn to phase 3 for our example program. Recall that the modulus for program point 6 equals 336. Accordingly, we determine the least solutions of the constraint systems $\mathcal{E}^\sharp_{\mathbb{Z}_{336}}, \mathcal{C}^\sharp_{\mathbb{Z}_{336}}$. We start with the already obtained sets of generators — modulo 336. In order to obtain a subsumption test for $\mathcal{E}^\sharp_{\mathbb{Z}_{336}}$ at variable q, we bring the set of matrices $\{Q_0, Q_1, Q_2\}$ computed in example 1 into echelon form (modulo 336). In our case this results in the matrices:

$$Q'_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad Q'_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 14 & 0 \\ 0 & 18 & 0 \end{bmatrix} \quad Q'_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 6 & 0 \end{bmatrix}$$

Propagating, e.g., the matrix Q_2 for the call at the edge $(1, 2)$, we obtain:

$$Q_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 15 & 0 \\ 0 & 192 & 1 \end{bmatrix}$$

Matrix Q_3 is already subsumed by the Q'_i. The same also holds for the propagation of the matrices Q_0 and Q_1. Therefore, the set $\{Q_0, Q_1, Q_2\}$ already represents the fixpoint. Accordingly, the module $\mathcal{C}^\sharp_{\mathbb{Z}_{336}}(6)$ is generated from the vectors:

$$z_1' = [1, 2, 0]^t \qquad z_2' = [1, 30, 36]^t \qquad z_3' = [1, 114, 228]^t$$

Next, we determine the module of valid equalities modulo 336 as the set of solutions of the following homogeneous system of equations over $\mathbb{Z}_{336}$:

$$[\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2] \cdot \begin{bmatrix} 1 & 1 & 1 \\ 2 & 30 & 114 \\ 0 & 36 & 228 \end{bmatrix} = [0, 0, 0]$$

or, equivalently,

$$[\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2] \cdot \begin{bmatrix} 1 & 0 & 0 \\ 2 & 28 & 0 \\ 0 & 36 & 84 \end{bmatrix} = [0, 0, 0]$$

The module of solutions is generated by the two vectors:

$$[312, 12, 0], \quad [0, 0, 28]$$

This corresponds to the congruence equations:

$$312 \cdot \mathbf{x}_0 + 12 \cdot \mathbf{x}_1 \equiv 0 \ [336] \qquad 28 \cdot \mathbf{x}_2 \equiv 0 \ [336]$$ □

Remark that all calculations on vectors or matrices in the third phase of the algorithm are in fixed modular rings and thus do not incur extra swells of intermediate numbers. In particular, we can use the complexity bounds from figure 2, to estimate the number of arithmetic and generalized gcd computations. For the intraprocedural case, we thus obtain $\mathcal{O}(n \cdot k^3 \cdot \log(m(X)))$ operations. Since the length $\log(m(X))$ of $m(X)$ is polynomially bounded in n and k, we obtain a polynomial algorithm.

In the interprocedural case, the number of operations is bounded by $\mathcal{O}(n \cdot k^8 \cdot \log(m(X))$. The modulus $m(X)$, though, can have exponential length. Therefore, we obtain an exponential complexity bound as stated in assertion 2. □

A subtle point in the algorithm over $\mathbb{Q}$ or $\mathbb{Z}$ is the potential swell of intermediate numbers. Our complexity analysis reveals that the total run-time of the interprocedural algorithm is polynomial in the size n of the program, the number k of variables and $\log(\Delta_{\mathcal{E}})$. Thus, the algorithm performs well if $\Delta_{\mathcal{E}}$ is found to be moderate. At the expense of loss of precision, this can always be enforced. Assume we have given us a threshold Δ. Whenever a matrix A with entry $|A_{ij}| > \Delta$ is to be added to some fixpoint variable, we instead add matrices $A^{(0)}, A^{(1)}$ which are obtained from A by replacing the too large entry with 0 and d, respectively, for some divisor d of A_{ij} (e.g., 1).

6 Guards

The draw-back of the interprocedural analyses of section 4 is that conditional branching is abstracted by non-deterministic choice. A natural class of guards to be taken into account are *equality guards* of the form $g = 0$ for $g \equiv g_0 + g_1\mathbf{x}_1 + \ldots + g_k\mathbf{x}_k$. In presence of equality guards, however, already the problem of determining at a given program point whether a variable always equals 0 is undecidable [12]. This holds even in absence of procedures. Accordingly, any effective analysis of programs with guards must be approximate. Intraprocedurally, an approximative treatment of equality guards has been considered both by Karr for fields [9] and by Granger for $\mathbb{Z}$ [5]. In both cases,

the effect of such a guard amounts to intersection of affine spaces. This idea also works for R-modules of extended states and any ring R:

$$[\![g = 0]\!]\, M \;=\; \langle M \cap \{[x_0, \ldots, x_k]^{\mathrm{t}} \mid x_0 = 1, \textstyle\sum_{j=0}^{k} g_j x_j = 0\}\rangle_{\mathsf{R}}$$

Computing the intersection can be reduced to solving a pair of linear equations: Assume $M = \langle G\rangle_{\mathsf{R}}$ where G is a finite set of generators. Let V denote a matrix containing the vectors of G as column vectors, let b denote the 0-th row of V. Then we obtain a system of generators for $[\![g = 0]\!]\, M$ by solving the system:

$$(g' V) \cdot \mathbf{y} = \mathbf{0} \qquad\qquad b \cdot \mathbf{y} = 1$$

for the row vectors $g' = [g_0, \ldots, g_k]$ and $b = [b_1, \ldots, b_q]$ and a column vector $\mathbf{y} = [\mathbf{y}_1, \ldots, \mathbf{y}_q]^{\mathrm{t}}$ of variables.

It is not obvious, though, how intersections can be lifted to the transformer level. Therefore, we suggest to *postpone* the decision taken at the guard. Instead of performing the intersection, we *accumulate* the value of the guard expression in an *indicator variable*. More precisely, assume that the edges with guards are numbered $k+1, \ldots, m$. Then we *instrument* the original program by introducing fresh variables $\mathbf{x}_{k+1}, \ldots, \mathbf{x}_m$, one for each guard. Initially, all these variables are assumed to have values 0. At the j-th guard $g = 0$, we place the assignment $\mathbf{x}_j := \mathbf{x}_j + g$. This corresponds to the matrix:

$$\left[\begin{array}{c|c|c|c} \multicolumn{1}{c|}{I_{k+1}} & \multicolumn{3}{c}{0} \\ \hline 0 & I_{j-k-1} & 0 & 0 \\ \hline g_0 \ldots g_k & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & I_{m-j} \end{array}\right]$$

The extra values stored in the indicator variables are then used for an improved treatment of calls in the constraint system $\mathcal{C}^\sharp_{\mathsf{R}}$. As an invariant, we insist in $\mathcal{C}^\sharp_{\mathsf{R}}$ that all indicator variables have values 0, since this is the case for all program runs permitted by the guards. Thus the first constraint now reads:

$$[\mathcal{C}^\sharp_{\mathsf{R}}1] \quad \mathcal{C}^\sharp_{\mathsf{R}}(\mathbf{Main}) \sqsupseteq \mathsf{R}^{k+1} \times \{0^{m-k}\}$$

Accordingly, we modify the constraints for calls to:

$$[\mathcal{C}^\sharp_{\mathsf{R}}6] \quad \mathcal{C}^\sharp_{\mathsf{R}}(v) \supseteq \langle \mathcal{E}^\sharp_{\mathsf{R}}(q)(\mathcal{C}^\sharp_{\mathsf{R}}(u)) \cap (\{1\} \times \mathsf{R}^k \times \{0^{m-k}\})\rangle_{\mathsf{R}} \qquad \text{if edge } (u, v) \text{ calls } q$$

Thus, having applied the transformations from $\mathcal{E}^\sharp_{\mathsf{R}}(q)$, we select just those vectors from the result whose indicator variables all equal 0. These can be determined by solving an appropriate system of linear equations. Altogether, we obtain for every effective PIR R, an enhanced interprocedural analysis which deals with equality guards and conservatively extends the corresponding intraprocedural analysis. In particular, this technique extends the known methods for fields, for modular rings $\mathbb{Z}_m$ and also for $\mathbb{Z}$.

The separation of computing valid affine relations from computing valid modular relations as in section 5 also returns sound information. In presence of guards, however, the latter may result in an extra loss of precision. Consider, e.g., the guard $8 - \mathbf{x}_1 = 0$. Assume that before the guard, we have the extended state $x = [1, 3]^{\mathrm{t}}$. Since $8-3 = 5 \neq 0$, x does not pass the guard both in an analysis over $\mathbb{Q}$ and over $\mathbb{Z}$. Assume, however, that we perform the third stage of the algorithm modulo 5. Since x satisfies the guard modulo 5, x is propagated through the guard — thus incurring an extra loss in precision.

7 Conclusion

We have provided a general framework for analyzing interprocedurally valid affine relations over any principal ideal ring R. In absence of guards, the analyses could be shown to be complete, i.e., to infer all valid relations of the given form. In particular, our framework covers the known cases of fields $\mathbb{Q}$ or $\mathbb{Z}_p$ (p a prime) as well as modular rings $\mathbb{Z}_m$ (m composite) and also provides an interprocedural extension of Granger's analysis of linear congruence relations. In order to obtain a faster analysis, we then decomposed the latter analysis into several instances of our framework. This new algorithm has the advantage that its run-time complexity can be explicitly determined. In particular, its intraprocedural variant runs in polynomial time. Finally, we indicated how the proposed techniques can be enhanced to deal interprocedurally with equality guards.

A key issue in designing efficient algorithms has been to bound potential swell of intermediately occurring numbers. In case of linear congruence analysis, we therefore refrained from computing the $\mathbb{Z}$-affine abstraction of the collecting semantics directly. Instead, we resorted to computations over modular rings. Remark that instead of performing a separate analysis for each program point X of interest we could as well perform one joint analysis using the lcm of the moduli for the X. The disadvantage, however, is that lengths of occurring numbers could then again grow unacceptably.

In order to keep the presentation simple, we have considered parameterless procedures and global variables only. Local variables, call-by-value passing of parameters and return values can be handled along the lines of [13]. At the expense of an increase in the complexity, our methods can also be used to determine valid *polynomial* relations up to a fixed degree d [12,13]. Further questions remain. It is still open whether it is possible to determine all valid *polynomial* relations — independent of a given degree bound. Also, it is desirable to design interprocedural analyses that deal precisely with further arithmetic operators.

References

1. G. Balakrishnan and T. W. Reps. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction, 13th Int. Conf. (CC)*, 5–23. LNCS 2985. Springer-Verlag, 2004.
2. B. Blanchet, P. Cousot, R. Cousot, J. Feret, C. Mauborgue, D. Mormiaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Int. ACM Conf. on Programming Language Design and Implementation (PLDI)*, 196–207, 2003.
3. N. Dor, M. Rodeh, and M. Sagiv. Cleanness Checking of String Manipulations in C Programs via Integer Analysis. In *8th Int. Static Analysis Symposium (SAS'01)*, 194–212. LNCS 2126, Springer Verlag, 2001.
4. P. Granger. Static Analysis of Arithmetical Congruences. *Int. J. of Computer Math.*, 165–190, 1989.
5. P. Granger. Static Analysis of Linear Congruence Equalities among Variables of a Program. In *Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT)*, 169–192. LNCS 493, Springer-Verlag, 1991.
6. S. Gulwani and G. Necula. Discovering Affine Equalities Using Random Interpretation. In *30th ACM Symp. on Principles of Programming Languages (POPL)*, 74–84, 2003.
7. S. Gulwani and G. Necula. Precise Interprocedural Analysis Using Random Interpretation. In *32th Ann. ACM Symp. on Principles of Programming Languages (POPL)*, 324–337, 2005.

8. J. Hafner and K. McCurley. Asymptotically Fast Triangularization of Matrices over Rings. *SIAM J. of Computing*, 20(6):1068–1083, 1991.
9. M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.
10. S. Lang. *Algebra, Third Edition.* Pearson Education, Inc., 1993.
11. A. Miné. Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors. In *European Conf. on Programming (ESOP)*, 3–17. LNCS 2986, Springer Verlag, 2004.
12. M. Müller-Olm and H. Seidl. A Note on Karr's Algorithm. In *31st Int. Coll. on Automata, Languages and Programming (ICALP)*, 1016–1028. Springer Verlag, LNCS 3142, 2004.
13. M. Müller-Olm and H. Seidl. Precise Interprocedural Analysis through Linear Algebra. In *31st ACM Symp. on Principles of Programming Languages (POPL)*, 330–341, 2004.
14. M. Müller-Olm and H. Seidl. Analysis of Modular Arithmetic. In *European Symposium on Programming (ESOP)*, 46–60. Springer Verlag, LNCS 3444, 2005.
15. A. Simon and A. King. Analyzing String Buffers in C. In *Algebraic Methodology and Software Technology, 9th Int. Conf. (AMAST)*, 365–379. LNCS 2422, Springer Verlag, 2002.
16. A. Storjohann. *Algorithms for Matrix Canonical Forms*. PhD thesis, ETH Zürich, Diss. ETH No. 13922, 2000.
17. A. Storjohann. *A Fast, Practical, and Deterministic Algorithm for Triangularizing Integer Matrices*. Tech. Rep. 255, ETH Zürich, 1996.
18. O. Zariski and P. Samuel. *Commutative Algebra, Vol. I.* Nostrand, Princeton, NJ, 1958.

A Proof of Theorem 1

The proof of statement (1) is a refinement of Granger's argument for computing a set of congruence relations characterizing $\langle G \rangle_{\mathbb{Z}}$. Let $V \in \mathbb{Z}^{r \cdot q}$ denote the matrix whose column vectors are the vectors from G. Then $x \in \langle G \rangle_{\mathbb{Z}}$ iff $V\,y = x$ for some (column) vector $y = [y_1, \ldots, y_q]^{\mathrm{t}} \in \mathbb{Z}^q$. Since V is linearly independent, we can find a unimodular matrix $T \in \mathbb{Z}^{r^2}$ such that $T \cdot V = \left[\frac{D}{0}\right]$ where D is an upper triangular $(q \times q)$-matrix and the product of the diagonal elements equals $\mathsf{det}(G)$ and thus divides m. In particular, $V\,y = x$ iff $(T \cdot V)\; y = T\; x$. In this matrix equation, the last $r - q$ rows constitute linear equations over $\mathbb{Z}$ whereas the first q rows can equivalently be formulated using linear equations modulo m. In order to see this, let d_i denote the i-th diagonal element of D and t_i the i-th row of T. Then the q-th row of the equation reads $d_q y_q = t_q \cdot x$ which is equivalent to the linear congruence equation $t_q \cdot x \equiv 0\; [d_q]$. By multiplying the remaining rows with d_q and subtracting suitable multiples of the q-th row, we can remove the q-th column of the remaining left-hand side of the equation system which leaves us with a similar problem where q has been decreased by one. Thus, we successively construct linear congruences with moduli $d_i \cdot \ldots \cdot d_q$ for $i = q$ down to $i = 1$. By scaling these equations with the products $p_i = \frac{m}{\mathsf{det}(G)} \cdot d_1 \cdot \ldots \cdot d_{i-1}$, we obtain equivalent congruences modulo m which together with the $m + 1 - q$ linear equations characterize all $x \in \langle G \rangle_{\mathbb{Z}}$.

Example 4. Consider the set $G = \{[2, 16, 34]^{\mathrm{t}}, [-2, -11, -24]^{\mathrm{t}}\}$. Let V denote the corresponding (3×2)-matrix of coefficients. Then there is unimodular matrix T with:

$$T = \begin{bmatrix} -7 & 1 & 0 \\ -8 & 1 & 0 \\ -1 & -2 & 1 \end{bmatrix} \qquad \text{and} \qquad V' = T \cdot V = \begin{bmatrix} 2 & 3 \\ 0 & 5 \\ 0 & 0 \end{bmatrix}$$

From the last row of T we thus can read off the linear equation:

$$-\mathbf{x}_0 - 2\mathbf{x}_1 + \mathbf{x}_2 \;=\; 0$$

The first two rows of the matrix equation $V'\,[y_1, y_2]^{\mathrm{t}} = T\,[x_0, x_1, x_2]^{\mathrm{t}}$ give us:

$$\begin{aligned} 2y_1 + 3y_2 &= -7x_0 + x_1 \\ 5y_2 &= -8x_0 + x_1 \end{aligned}$$

Subtracting three times the second equation from 5 times the first one gives us:

$$\begin{aligned} 10y_1 \qquad &= -11x_0 + 2x_1 \\ 5y_2 &= \;\;-8x_0 + \;\; x_1 \end{aligned}$$

This provides us with the following two congruence equations which together with the linear relation characterize the $\mathbb{Z}$-module generated by G:

$$\begin{aligned} -11\mathbf{x}_0 + 2\mathbf{x}_1 &\equiv 0\ [10] \\ -8\mathbf{x}_0 + \;\; \mathbf{x}_1 &\equiv 0\ \;\,[5] \end{aligned}$$

□

It remains to consider statement 2. The case $m' = 0$ is trivial. So let $m' > 0$. As the linear congruence equation $b' \cdot \mathbf{x} \equiv 0\ [m']$ is satisfied for a vector $v \in \mathbb{Z}^r$ iff $(d' \cdot b') \cdot \mathbf{x} \equiv 0\ [h]$ is satisfied for v (recall that $h = m' \cdot d'$), it suffices to show that $\langle E_0 \cup \{d \cdot b \mid b \in E_m\}\rangle_{\mathbb{Z}_h}$ characterizes the linear congruence relations valid for all vectors in G modulo h. Thus we show: $b' \cdot \mathbf{x} \equiv 0\ [h]$ is satisfied by all vectors in G iff $b' \in \langle E_0 \cup \{d \cdot b \mid b \in E_m\}\rangle_{\mathbb{Z}_h}$.

First of all, if $b' \in E_0$, then $b' \cdot x = 0$ and hence also $b' \cdot x = 0\ [h]$ for all $x \in G$. Moreover if $b \in E_m$ then $b \cdot x = 0\ [m]$ and hence $(d \cdot b) \cdot x = 0\ [h]$ for all $x \in G$ because $h = d \cdot m$. Thus, for any $b' \in \langle E_0 \cup \{d \cdot b \mid b \in E_m\}\rangle_{\mathbb{Z}_h}$, $b' \cdot \mathbf{x} \equiv 0\ [h]$ is satisfied by all vectors in G because validity of linear congruence relations is preserved by linear combinations. This shows the "if"-direction.

For the "only if"-direction, let again $V \in \mathbb{Z}^{r \cdot q}$ be the matrix whose columns are formed by the vectors from G. Note that for any $l > 0$ and $b \in \mathbb{Z}_l^r$, the linear congruence relation $b \cdot \mathbf{x} \equiv 0\ [l]$ holds for all $x \in G$ iff b is a solution of the following equation system E over $\mathbb{Z}_l$: $\mathbf{y} \cdot V = \mathbf{0}$. Similarly, for $b \in \mathbb{Z}^r$ the relation $b \cdot \mathbf{x} \equiv 0$ is satisfied by all vectors in G if b is a solution of the equation system E over $\mathbb{Z}$. Let b' be a solution of E over $\mathbb{Z}_h$. We need to show that $b' = b_0' + d \cdot b_1'$ where b_0' is a solution of E over $\mathbb{Z}$ and b_1' is a solution over $\mathbb{Z}_m$. As the columns of V are linearly independent, we can construct a unimodular matrix T such that $V' = T \cdot V = \left[\frac{D}{0}\right]$ where D is upper triangular with diagonal elements $d_1, \ldots, d_q$ and $\mathsf{det}(G) = d_1 \cdot \ldots \cdot d_q$ divides m. Now we consider the homogeneous system E': $\mathbf{y} \cdot V' = \mathbf{0}$. The vector $b'' = b' \cdot T^{-1}$ is a solution of E' over $\mathbb{Z}_h$. We can write b'' in the form $b_0'' + b_1''$ where all components $i = 1, \ldots, q$ of b_0'' and all components $i = q+1, \ldots, r$ of b_1'' are 0. By inspecting E', we see that b_0'' is also a solution of E' over $\mathbb{Z}$ and b_1'' is also a solution over $\mathbb{Z}_h$. By induction for $i = q$ down to $i = 1$, we verify in addition that the i-th entry of b_1'' equals 0 modulo $d \cdot d_{i+1} \cdot \ldots \cdot d_q$. Thus, $b_1'' = d \cdot y$ for some $y \in \mathbb{Z}^r$. Since $d \cdot y \cdot V' = b_1'' \cdot V' \equiv \mathbf{0}\ [d \cdot m]$, we conclude that also $y \cdot V' \equiv \mathbf{0}\ [m]$. Therefore, y is a solution of the system $\mathbf{y} \cdot V' = \mathbf{0}$ over $\mathbb{Z}_m$. Now, we choose $b_0' = b_0'' \cdot T$ and $b_1' = y \cdot T$ such that $b' = b'' \cdot T = (b_0'' + d \cdot y) \cdot T = b_0' + d \cdot b_1'$. Moreover, we have $b_0' \cdot V = b_0'' \cdot T \cdot V = b_0'' \cdot V'$ such that b_0' solves equation system E over $\mathbb{Z}$ because b_0'' solves E' over $\mathbb{Z}$. Similarly, $b_1' \cdot V = b_1'' \cdot T \cdot V = b_1'' \cdot V'$ such that b_1' solves E over $\mathbb{Z}_m$ because b_1'' solves E' over $\mathbb{Z}_m$. This completes the proof. □

Finding Basic Block and Variable Correspondence

Iman Narasamdya and Andrei Voronkov

University of Manchester
{in, voronkov}@cs.man.ac.uk

Abstract. Having in mind the ultimate goal of translation validation for optimizing compilers, we propose a new algorithm for solving the problem of finding basic block and variable correspondence between two (low-level) programs generated by a compiler from the same source using different optimizations. The essence of our technique is interpretation of the two programs on random inputs and comparing the histories of value changes for variables. We describe an architecture of a system for finding basic block and variable correspondence and provide experimental evidence of its usefulness.

1 Introduction

Verifying the optimizing phase of a compiler has become crucial as developers have been relying on this phase to produce high performance code. However, proving the correctness of the optimizing phase is infeasible due to its size, its sophisticated algorithms and data structures, as well as ongoing evolution and modification. *Translation validation* [7] is an alternative but feasible approach to compiler correctness, which can be applied to the optimizing phase [6,10,8]. The idea of translation validation is as follows: instead of proving the correctness of the optimizing phase for *every* possible program, prove *for a single program* that the program and its optimized version are semantically equivalent.

In this paper we take the following view of translation validation. We have two programs P and P', and each of them is a result of compiling the same source program, but unlike P, the compilation of P' involves the optimizing phase. Both programs are written in the same intermediate language. We call the program P the *original program*, and the program P' the *optimized program*.

Knowing that a variable in P corresponds to a variable in P' gives us a valuable information that can be used to prove the equivalence of P and P' automatically. Intuitively, a variable x_1 in the original programs corresponds to a variable x_2 in the optimized program if they have the same values at some control blocks for all possible runs of the two programs on the same input values. We shall formulate the right notion of correspondence in a formal way later.

In this paper *block* is a sequence of instructions that is always entered at the beginning and exited at the end. With this definition, we consider a *program point* as a block consisting of one instruction. A block is called a *basic block* if the sequence of instructions is maximal.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 251–267, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Consider the following simple programs:

P :	P' :
w_1 := n;	x_1 := n;
while $w_1 > 0$ **do**	**while** $x_1 > 0$ **do**
w_2 := $w_1 - 1$;	x_1 := $x_1 - 2$;
w_1 := $w_2 - 1$;	**call** $f(x_1)$
call $f(w_1)$	**od**
od	

In these programs n is an argument and w_1, w_2, x_1 are local variables. Suppose that the function f does not have any side effect. If we can establish that the variable w_1 in P corresponds to the variable x_1 in P', then we can verify that the two programs are equivalent without generating loop invariants. Indeed, using this information we can check that the two programs will perform the same sequence of calls of $f(\ldots)$ by the following kind of reasoning. First, the values of w_1 and x_1 coincide at the entries of the two loops. Second, if they coincide at some iteration of the loops, they also coincide at the next iteration. Third, if they coincide at some iteration of the loops, the function f will be called with the same arguments in both programs. Finally, if the loop exit condition $w_1 \leq 0$ is satisfied in P, the loop exit condition $x_1 \leq 0$ is also satisfied in P'.

Note that this reasoning also *proves* that w_1 in P and x_1 in P' correspond to each other. However, before performing this reasoning we must in some way *guess* that w_1 in the original program corresponds to x_1 in the optimized program. Moreover, we also have to establish some correspondence between control blocks in the two programs: blocks in which the corresponding variables always have the same values. This paper deals with "guessing" a basic block and variable correspondence. We do not yet consider the VC generation from a given correspondence or proving the VCs.

By only knowing that one program is an optimized version of the other, it is not trivial to construct automatically a basic block and variable correspondence. Optimizations can change the structure of a program, for instance, while-do loops are transformed to do-while loops to be able to move loop invariants. Optimizations might also include eliminating existing branches and introducing new branches to the program. In this paper we do not try to address any *reordering transformation*, that is any transformation that changes the order of execution of code, without adding or deleting any executions of any statement [5].

This paper proposes a new technique in constructing a basic block and variable correspondence between P and P'. The idea of this new technique is to execute P and P' separately with the same initial store, also called a *memory state* here. The values stored in the memory are generated randomly upon demand. For example, when a program accesses an uninitialized memory location, we can create a new piece of memory and fill it with a random value. Furthermore, while executing the programs, the values assigned to each variable and the blocks in which these assignments occur are recorded. If the sequences of *value changes* of two variables are the same, then the variables probably correspond to each other, and the block in which the changes occur might also correspond to each other.

The problem of finding a basic block and variable correspondence between P and P' is a hard problem. No single technique is able to cover all possible optimizations

applied to the source program. The emphasis of our work here is to develop a *cheap* technique that could help to find a basic block and variable correspondence. This correspondence in turn can help us generate a verification condition which is sufficient to prove the equivalence of P and P'. Our technique is considerably cheap for the following reasons. First, our technique amounts to building an interpreter to perform program executions. As the language in which P and P' are written is usually simple, the interpreter is easy to develop. Moreover, it does not take a sophisticated algorithm to determine the sameness of value changes between two records. Another advantage of our new technique is that it needs only the code of the original and the optimized programs but no further information from the optimizing phase. Therefore, it can be applied to verify the optimizing phase of different compilers without instrumenting them any further.

The remainder of this paper is organized as follows. Section 2 gives an overview of some recent existing techniques in constructing basic block and variable correspondences. Section 3 states formally the problem of finding basic block and variable correspondence. In Section 4 the idea of the new technique is discussed. Afterwards, in Section 5, we discuss the syntax and semantics of an intermediate language used throughout this paper . Section 6 discusses a *randomized interpreter* used to evaluate programs written in the intermediate language. Finally, section 7 describes some experimental results. An extended version of this paper is available at `http://www.cs.man.ac.uk/~voronkov/sas_fullpaper.ps`.

2 Related Work

One technique related to translation validation is *Necula's technique* [6]. In this technique, each of the original and the optimized programs is firstly evaluated symbolically into a series of mutually recursive function definitions. A basic block and variable correspondence is inferred by a scanning algorithm that traverses the function definitions. For example, when the scanning algorithm visits a branch condition e in the original program, it determines whether e is eliminated due to the optimizations. If it is eliminated, then the information collected is either $e = 0$ or $\neg e = 0$, depending on which branch of e is preserved in the optimized program. If e is not eliminated, then it corresponds to another branch condition e' in the optimized program. The information collected is either $e = e'$ or $e = \neg e'$, depending on the correspondence of e's and e''s branches. This shows that, besides symbolic evaluation, Necula's technique has to solve some equalities to determine which branches are eliminated and also to determine the correspondence between branches in the two programs. Moreover, to find a basic block correspondence Necula's technique uses some heuristics which are specific to the *GNU C compiler*. This limits the applicability of Necula's technique to verifying other compilers.

Another translation validation technique is *VOC* [11]. We overview VOC for structure preserving transformations only. Such transformations admit a mapping between some program points in P and P'. In VOC a basic block and variable correspondence is represented by a mapping from some blocks in P' to some blocks in P, and also by a data abstraction. The domain and range of the block mapping form sets of *control*

blocks. VOC chooses the first block of each loop body as a control block. The data abstraction is constructed as follows. For each block B_i in P', and for every path from block B_j leading to B_i, a set of equalities $v = V$ is computed, where v and V are variables in P and P' respectively. The equalities are implied by invariants reaching B_j, transition system representing the path from B_j to B_i and its counterpart in P, and the current constructed data abstraction. This requires the implementation of VOC to use a prover to generate a data abstraction. Moreover, an implementation of VOC for *Intel's ORC compiler*, *VOC-64*, tries the variable equalities for every pair of variables except for the temporaries introduced by the compiler. This trial is performed by scanning the symbol table produced by the compiler [2]. However, not every compiler provides the symbol table as a result of compilation, thus this limits the applicability of VOC-64.

A quite recent translation validation technique is *Rival's technique* [9]. The technique provides a unifying framework for the certification of compilation and of compiled programs. Similarly to Necula's technique, the framework is based on a symbolic representation of the semantics of the programs. Rival's technique extracts basic block and variable correspondence from the standard debugging information if no optimizations are applied. However, when some optimizations are involved in the compilation, the optimizing phase has to be instrumented further to debug the optimized code and generate the correspondence between the original and the optimized programs. One technique to automatically generate such a correspondence is due to *Jaramillo* et. al [4]. In this technique, the optimized programs initially starts as an identical copy of the original one, so that the mapping starts as an identity. As each transformation is applied, the mapping is changed to reflect the effects of the transformation. Thus, in this technique, one needs to know what and in which order the transformations are applied by the optimizing phase.

3 Basic Block and Variable Correspondence

In this section we formalize the problem we are trying to solve. We will only be dealing with *programs* divided into blocks. A concrete notion of program will be defined later in Section 5. We assume that every program defines a transition relation with two kinds of transition: (i) transitions $(\beta_1, \sigma_1) \to (\beta_2, \sigma_2)$; (ii) transitions $(\beta_1, \sigma_1) \to \sigma_2$, where β_1, β_2 are blocks and σ_1, σ_2 are stores. Intuitively, the second kind of transition brings the program to a terminal state. The *run* of such a program is either an infinite sequence $(\beta_0, \sigma_0), (\beta_1, \sigma_1), \ldots$ or a finite sequence $(\beta_0, \sigma_0), (\beta_1, \sigma_1), \ldots, (\beta_n, \sigma_n), \sigma_{n+1}$. Here $\beta_0, \beta_1, \ldots$ is the sequence of blocks visited in this run and σ_i is the store when the run reaches β_i.

Let $\bar{b}$ be a sequence of distinct blocks in P and R be a run. Denote by $R|_{\bar{b}}$ the subsequence of R consisting of the blocks occurring in $\bar{b}$.

Let P and P' be two programs, $\bar{b} = b_1, \ldots, b_k$ be a sequence of distinct blocks in P and $\bar{b}' = b'_1, \ldots, b'_k$ be a sequence of distinct blocks in P' of the same length. Let also $\bar{x} = x_1, \ldots, x_m$ be a sequence of variables[1] in P and $\bar{x}' = x'_1, \ldots, x'_m$ be a sequence of variables in P', also of the same length. In the sequel we will refer to $\bar{b}$ and $\bar{b}'$ as *control blocks* and to $\bar{x}$ and $\bar{x}'$ as *control variables*.

[1] For simplicity, we consider variables as memory locations.

We say that there is a *block and variable correspondence* between $(\bar{b}; \bar{x})$ and $(\bar{b}'; \bar{x}')$ if, for every pair of runs $R = (\beta_0, \sigma_0), (\beta_1, \sigma_1), \ldots$ and $R' = (\beta'_0, \sigma'_0), (\beta'_1, \sigma'_1), \ldots$ of the programs P and P', respectively, on the same inputs and the same initial store, (that is, $\beta_0 = \beta'_0$ and $\sigma_0 = \sigma'_0$) the following conditions hold. Let

$$R|_{\bar{b}} = (\beta_{i_0}, \sigma_{i_0}), (\beta_{i_1}, \sigma_{i_1}), \ldots \qquad R'|_{\bar{b}'} = (\beta'_{j_0}, \sigma'_{j_0}), (\beta'_{j_1}, \sigma'_{j_1}), \ldots$$

Then $R|_{\bar{b}}$ and $R'|_{\bar{b}'}$ have the same length and for every non-negative integer n the following conditions hold:

1. $\beta_{i_n} = b_\ell$ if and only if $\beta'_{j_n} = b'_\ell$, for all ℓ;
2. $\sigma_{i_n}(x_1) = \sigma'_{j_n}(x_1), \ldots, \sigma_{i_n}(x_m) = \sigma'_{j_n}(x_m)$;
3. $\sigma_{i_{n+1}}(x_1) = \sigma'_{j_{n+1}}(x_1), \ldots, \sigma_{i_n+1}(x_m) = \sigma'_{j_n+1}(x_m)$.

That is, in R and R' the control blocks are visited in the same order, and the values of the control variables at the entries and exits of the visited control blocks are the same.

Our main goal is to find, in a fully automatic way, a correspondence between program points and variables of P and P'. Note that we always have a correspondence when $\bar{b}$ is an empty sequence. Likewise, we always have a correspondence when $\bar{x}$ is an empty sequence. As a consequence, there is no largest correspondence. However, we are interested in correspondences in which $\bar{b}$ is "as large as possible", and similarly for $\bar{x}$.

The definition of basic block and variable correspondence above allows us to trade variable correspondence for block correspondence and vice versa. Consider the following programs with n as their arguments:

```
Program P:                         Program P':
b0 :  if  n ≤ 0 then               b0' :  if  n > 0 then
        b2 :  x1  := n                      b2' :  x2'  := 1
              x2  := 0                     else
      else                                  b3' :  x2'  := 0
        b3 :  x1  := n             b4' :  x1'  := n
              x2  := 1                    x3'  := x2'
b4 :  x1  := n
      x3  := x2
```

The program P' can be obtained by applying dead code elimination to P. If we can establish that x_1, x_2, and x_3 in P correspond to their primed counterparts in P', we could only construct a block correspondence between b_0 and b'_0, and also between b_4 and b'_4. The block b_2 does not correspond to the block b'_3 since the values of x_1 and x'_1 after executing these blocks are different. When we sacrifice the correspondence between x_1 and x'_1, we obtain a larger block correspondence, that is between b_2 and b'_3, and also between b_3 and b'_2. The resulting block correspondence is crucial if we have to establish a branch correspondence.

We can introduce variations on the basic block and variable correspondence problem. For example, if a variable is initialized inside a block, we can restrict the definition to its value at the block exit only. We can change the definition so that a single block in one of the programs will correspond to several blocks in another program. This will help us to cope with such optimizations as *loop invariant hoisting*. Likewise, we can

consider the *single static assignment* (or *SSA*) [1] form of programs in which a variable may change its value only inside a single basic block. The technique we discuss in this paper is equally applicable to these modifications.

4 Random Interpretation

In this section we introduce the technique of *random interpretation* that allows one to address the block and variable correspondence problem. The idea of the technique is to evaluate both the original program and its optimized version separately with the same initial randomly generated memory state. A memory state can be thought of as a function mapping memory locations to the content of the memory at these locations. While evaluating each program, we record the values assigned to each variable and the program points at which the assignments occur. This record forms a *history* of values assigned to a variable. Let us define the notion of history formally.

As usual, we say that a block b *defines* a variable x if b contains an assignment to x. Consider a run R of a program P and let x be a variable occurring in P. Let $\bar{b}$ be the set of all blocks in P defining x and $R|_{\bar{b}} = (\beta_0, \sigma_0), (\beta_1, \sigma_1), \ldots$. Then $\beta_0, \beta_1, \ldots$ are the only blocks in this run which may change the value of x. We call the *history of x in R* the sequence of pairs

$$(v_0, \beta_0), (v_1, \beta_1), \ldots \tag{1}$$

where each v_i is the value of x *at the exit* of β_i. Now, given the history h_x of x in R of the form (1) we call the *value change sequence of x in R* any subsequence of (1)

$$(v_{j_0}, \beta_{j_0}), (v_{j_1}, \beta_{j_1}), \ldots$$

that can be obtained from (1) by repeated applications of the following transformation: if the sequence contains two consecutive pairs with the same value:

$$\ldots, (v, \beta_i), (v, \beta_{i+1}), \ldots,$$

then remove either (v, β_i) or (v, β_{i+1}) from it. This transformation is applied until there are no such pairs.

Let $h = (v_0, \beta_0), \ldots$ and $h' = (v'_0, \beta'_0), \ldots$ be two sequences of value changes. We write $h \lhd h'$ if the sequence of values $v_0, v_1, \ldots$ is a prefix of $v'_0, v'_1, \ldots$. In other words, the length of h is smaller than or equal to the length of h' and for all k such that (v_k, β_k) occur in h we have $v_k = v'_k$. We write $h \lhd\rhd h'$ if $h \lhd h'$ and $h' \lhd h$. We will use the same notation for histories. Let h and h' be two histories. Then we write $h \lhd h'$ if the relation $\lhd$ holds on the sequences of value changes corresponding to h and h', and similar for $\lhd\rhd$ in place of $\lhd$. For example if the history of a variable x in a run R is

$$h = (1, b_1), (2, b_2), (2, b'_2), (3, b_3), (4, b_4), (5, b_5),$$

then there are two value change sequences of x in R:

$$h_1 = (1, b_1), (2, b_2), (3, b_3), (4, b_4), (5, b_5) \text{ and}$$
$$h_2 = (1, b_1), (2, b'_2), (3, b_3), (4, b_4), (5, b_5).$$

Obviously $h \lhd \rhd h_1$ and $h \lhd \rhd h_2$ (the relation $\lhd\rhd$ always holds between a history and the value change sequence obtained from this history).

Consider another example. Assume that the histories of variables x and x' in runs of P and x' in P' with the same initial store are, respectively

$$h = (1, b_1), (2, b_2), (3, b_3), (4, b_4), (5, b_5), \text{ and}$$
$$h' = (1, b'_1), (2, b'_{2,1}), (2, b'_{2,2}), (3, b'_3), (4, b'_4), (5, b'_5).$$

Then we have $h \lhd \rhd h'$. This suggests that there may be a correspondence between $(b_1, b_2, b_3, b_4, b_5; x)$ and $(b'_1, b'_{2,1}, b'_3, b'_4, b'_5; x')$ and also between $(b_1, b_2, b_3, b_4, b_5; x)$ and $(b'_1, b'_{2,2}, b'_3, b'_4, b'_5; x')$.

We are going to use the notions of history and sequence of value changes in translation validation as follows:

1. Run an interpreter several times on P and P' on a randomly generated store, record histories of variables and *guess* a block and variable correspondence using the corresponding histories.
2. *Prove* that the guessed correspondence is, indeed, a correspondence;
3. *Using* this correspondence, *prove* the equivalence of P and P'.

This paper is only concerned with the first part of this process. Note that in the first part we only *guess* a correspondence, verification of this correspondence is not described here. Since we only guess a correspondence, we will refer to it in the sequel as a *guessed correspondence*.

On interpreting a program, if the content of a memory location is used without any prior initialization, then that memory location is initialized with a random value. This is the reason for calling this technique *random interpretation*. Moreover, since there is no guarantee that random interpretation terminates, we abort it after some number of steps.

We guess a correspondence by making several runs of the two programs and memorizing histories of variables. These histories are then compared using the relation $\lhd\rhd$ for terminated runs and $\lhd$ for aborted ones.

Of course, after guessing a correspondence the verification may fail, after which we can try to guess another correspondence. We think that our technique can nicely complement other existing techniques for the following reasons.

(i) The notion of correspondence (as formalized here, or similar notions) seems to be fundamental in all approaches to translation validation. If one wants to implement a validating compiler, then the compiler should produce a correspondence. However, translation validation of third-party compilers requires some way of finding a correspondence. (ii) Our technique for guessing a correspondence does not use symbolic evaluation or proofs and is, therefore, cheap. Moreover, the space of all possible correspondences is huge, while our techique normally guesses a reasonably-sized correspondence after a small number of runs. (iii) Other techniques can be combined with ours. For example, if each of the two programs contains a single copy of a call of the same function, we can require that every correspondence includes the blocks with the function calls. In general, one can combine random interpretation with a symbolic interpretation.

One can argue that the relationship between histories of variables and correspondences is loose. For example, one might argue that the same variable in the optimized program may correspond to several variables in the original one. To tackle this problem, the original program P and the optimized program P' are transformed into their SSA forms prior to interpreting them. In SSA form a block in which a variable can changes its value is uniquely identified by this variable, therefore the notion of correspondence can be simplified by using only sequences of variables. Moreover, in SSA form, instead of using value change sequences, we can simply consider the histories of variables to guess a variable correspondence.

5 The Intermediate Language *IL* and the Memory Model

The Syntax of *IL*. The original program and its optimized version are written in the same low-level intermediate language, which is subsequently called *IL*. This section discussed the syntax and semantics of this language.

Figure 1 describes the syntax of *IL*. Instructions consist of move instruction, jump instruction, conditional jump instruction, and return instruction. Every instruction has a unique label l_1, and all of them but jump have the label l_2 of the next instruction. We sometimes denote an instruction by its label. At this early stage, the *IL* language does not include function calls. Expressions in the *IL* language are of the following forms: integer constant, register[2], global variable, escaping memory location[3], binary arithmetic operation, and relational operation.

The Chunk Memory Model. Hitherto, a memory has usually been modelled as an infinitely long array. In this model its is assumed that memory locations for global variables do not overlap with each other and with other memory locations. Similarly to memory locations for registers. Moreover, each stack frame is represented by a finite sub-array of the infinitely long array.

To prove program equivalences, we sometimes have to provide evidence that updating a variable does not change the values of other variables. For instance, updating a global variable does not affect the values of other global variables, that is two global variables g_1 and g_2 never refer to the same memory location. In the *IL* language global variables g_1 and g_2 are written as $[g_1]$ and $[g_2]$, where g_1 and g_2 are memory addresses. In the above memory model, to provide evidence that the memory locations do not overlap, we have to show that there exist integers g_1 and g_2 such that for every integer n, $g_1 + n \neq g_2$. This statement is obviously false.

In this section a new memory model is proposed. One may think of this model as corresponding to the memory model of C. The model describes a memory as a collection of chunks. A *chunk* is a finite, contiguously allocated set of objects. In this memory model a register can be considered as a chunk of size one. Thus, there is no difference between registers and ordinary memory pieces. The values stored in chunks can either be constants or references to some chunks. Furthermore, in this paper we

[2] In this paper the notions of register and temporary are used interchangeably.

[3] An escaping memory location is a memory location whose address can be taken.

Instruction	i	$::= l_1 : \; mi\, l_2 \mid l_1 : \; \texttt{jump}(o) \mid l_1 : \; \texttt{cjump}(rel, l)\, l_2$ $\mid l_1 : \; ret$
Move Instruction	mi	$::= lval \leftarrow e$
Left Value	$lval$	$::= r \mid [a]$
Return	ret	$::= \texttt{return} \mid \texttt{return}\, r$
Expression	e	$::= o \mid rel$
Relational Operation	rel	$::= o_1 \; rel \; o_2$
Relational Operator	rop	$::= > \mid \geq \mid < \mid \leq \mid = \mid \neq \mid \ldots$
Arithmetic Operation	$arith$	$::= o_1 \; bop \; o_2$
Arithmetic Operator	bop	$::= + \mid - \mid * \mid / \mid \ldots$
Operand	o	$::= n \mid r \mid g \mid [a] \mid arith$
Address	a	$::= r \mid g \mid a + ao \mid ao + a \mid a - ao$
Address Offset	ao	$::= r \mid n \mid ao_1 \; bop \; ao_2$
Integer Constant	n	$::= \mathcal{Z}$
Register	r	$::= \texttt{r}_1 \mid \texttt{r}_2 \mid \ldots$
Global Name	g	$::= \langle identifiers \rangle$
Label	l	$::= \mathcal{N}$

Fig. 1. The intermediate Language *IL*

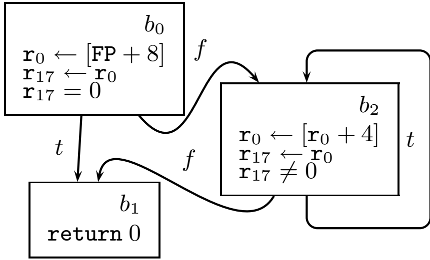


Fig. 2. The linked list traversal program of Example 1

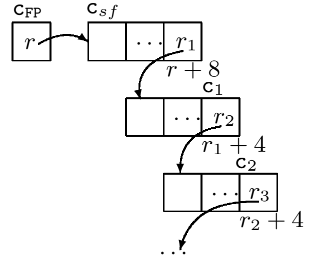


Fig. 3. The chunk-based memory after interpreting the program in Figure 2

focus on intra-procedural optimizations, and thus there is one chunk dedicated to representing the current stack frame. Memory locations on this particular chunk are often called *stack memory locations*. The new memory model is called *the chunk memory model*. This model can be used to provide evidence that some variables never refer to the same location. For example, such information can be provided by a language standard (pointers to two incompatible types cannot coincide, memory allocation operator always returns a new piece of memory etc.).

Example 1. Consider the program in Figure 2. Here, `FP` denotes the frame pointer. In a higher-level language the program can be viewed as a program that walks over a linked list. With this understanding, the memory selection $[\texttt{r}_0 + 4]$ denotes the next element of a node in the linked list.

The memory after interpreting the program is depicted in Figure 3. The chunk $\texttt{c}_{\texttt{FP}}$ represents the frame pointer, and the chunk $\texttt{c}_{sf}$ is the one dedicated to representing the current stack frame. The length of the chained chunks depends on

- the random initial value r_1 assigned to the chunk $\texttt{c}_{sf}$ at the address $r + 8$, where r is the random initial value for the frame pointer $\texttt{FP}$ and r_1 is a reference to the chunk $\texttt{c}_1$; and
- the random initial value r_{i+1} assigned to the chunk $\texttt{c}_i$ at address $r_i + 4$, where r_{i+1} is a reference to the chunk $\texttt{c}_{i+1}$, for $i > 1$.

Formal Semantics of *IL*. First we introduce a notion of values. Evaluation of an expression results in a value, which can either be an integer constant or a reference to a chunk. The following definition describes values formally:

$$\textit{Value } v \ ::= \ n \mid \textit{ref}; \ \textit{Reference ref} \ ::= \ (c, n); \ \textit{Chunk } c \ ::= \ \texttt{c}_1 \mid \texttt{c}_2 \mid \ldots$$

A *reference* is a pair (c, n), where c denotes the chunk to which the reference points and n denotes an index of the chunk. References are considered as memory location.

To describe the semantics, we introduce two operations on memory states. Denote by M, R, and V the sets of all memory states, references, and values, respectively. The functions

$$\texttt{sel} : M \times R \to V \ \text{ and } \ \texttt{upd} : M \times R \times V \to M$$

access (respectively, update) memory states. The value $\texttt{sel}(m, r)$ is the content of the memory state m at the address r, where r is a reference. The memory state $\texttt{upd}(m, r, v)$ is obtained from the memory state m by updating m at the address r with the value v.

To define the dynamic semantics of IL, we consider registers and names of global variables as references $(c, 0)$ for some chunk c. The association between them and such references is formalized using the notion of *static environment*. The environment consists of two partial injective functions: Env_n, which maps names of global variables and registers to references; and Env_l, which maps integers to instructions.

The dynamic semantics of *IL* is specified in terms of *operational semantics* given by a simultaneous inductive definition of the following relations:

$$\begin{array}{l} \textit{Instruction interpretation} : (m, i) \mapsto (m', i'); \\ \textit{Expression interpretation} : \ (m, e) \mapsto v. \end{array}$$

The instruction interpretation $(m, i) \mapsto (m', i')$ means the following: interpreting the instruction i with the memory state m yields a new memory state m', and the interpretation is followed by interpreting the instruction i' with the new memory state m'. Likewise for the expression interpretation, but the evaluation does not change the memory state.

Due to lack of space, we only show some rules in the inductive definition. First, memory can only be accessed through references. In Example 1 the evaluation of $\texttt{FP}+8$ in $[\texttt{FP} + 8]$ must result in a reference. The following rule describes this situations:

$$\frac{(m, a) \mapsto \textit{ref} \quad \texttt{sel}(m, \textit{ref}) = v}{(m, [a]) \mapsto v}$$

Arithmetic operations on references are only applied to the index. Adding two references and subtracting a reference from a constant do not make any sense. Thus, the rules describing binary operations have to distinguish the evaluation results of their operands, for example:

$$\frac{(m, o_1) \mapsto (c, n_1) \quad (m, o_2) \mapsto n_2}{(m, o_1 + o_2) \mapsto (c, n_1 + n_2)} \quad \frac{(m, o_1) \mapsto (c, n_1) \quad (m, o_2) \mapsto (c, n_2)}{(m, o_1 - o_2) \mapsto n_1 - n_2}$$

The right-hand side rule above specifies that subtracting a reference from another reference is allowed as long as both references point to the same chunk.

Rules for instructions can be described similarly. For example, assigning a value to a register is described by the following rule for move instructions:

$$\frac{Env_n(r) = ref \quad (m, e) \mapsto v \quad Env_l(l_2) = i}{(m, l_1 : \ r \leftarrow e\ l_2) \mapsto (\mathtt{upd}(m, ref, v), i)}$$

The memory location where the value is stored is obtained by looking up Env_n.

6 Randomized Interpreter

This section discussed a randomized interpreter that imitates the work of the interpreter except that the memory state is generated using random number generator.

The Algorithm of the Randomized Interpreter. Consider again the program in Figure 2. The content of memory location denoted by $\mathtt{FP} + 8$ is used but without any prior initialization, so the value of evaluating $[\mathtt{FP} + 8]$ for the first time is not known. To denote the unknown value resulting from expression evaluation, we extend the set of values by introducing an *undefined value* $\bullet$. Initially, when the randomized interpreter evaluates a program, every location in the memory state contains $\bullet$. The behavior of the randomized interpreter can be defined by extending the semantics of *IL* to work with the undefined value. However, the extension is not so straightforward for two reasons. First, reading a location containing $\bullet$ results in a change of memory since the location will be filled with a random generated value. Second, *IL* has no type information, so the randomized interpreter does not know if the location should be initialized with a constant or a reference.

The first problem is solved by changing the relation of expression evaluation to allow updating memory states, that is $(m, e) \mapsto_r (m', v)$. To solve the second problem, we introduce a new kind of value called *conditional value*. This kind of value is denoted by $(ref_1\ ?\ n : ref_2)$, which means that the *definite value* is n if the content of ref_1 was firstly initialized to a reference, or otherwise ref_2. Both n and ref_2 are often called the *definite forms* of the conditional value. Furthermore, we also introduce a function $rand$ that generates random pairs (ref, n), where ref is a reference to a new chunk. The content of a newly created chunk is $\bullet$ everywhere.

A conditional value might become definite at some point during an interpretation. For example, multiplication can only be applied to integer operands. Thus, in multiplying a conditional value $(ref\ ?\ n_1 : ref_1)$ with an integer n_2, the conditional value gets

definite to the integer n_1, and the content of ref is known to be firstly initialized with a reference. To capture this, we introduce a new operation on memory:

$$\mathtt{def} : M \times R \times \{0,1\} \rightarrow M.$$

The operation $\mathtt{def}(m, ref, b)$ returns a new memory state m' obtained from m by updating with a definite value $defv$ every reference ref' in m at which $\mathtt{sel}(m, ref')$ is $(ref \mathrel{?} defv_1 : defv_2)$ for some definite values $defv_1$ and $defv_2$, such that, if b is one, then $defv$ is equal to $defv_1$, otherwise $defv_2$.

Due to lack of space, we only show some rules describing the algorithm of the randomized interpreter. Suppose, on evaluating $[a]$ with a memory state m, a evaluates to a reference v_a, but the memory selection $\mathtt{sel}(m, v_a)$ yields the undefined value. The interpreter then generates a random conditional value as described by the following rule:

$$\frac{(m, a) \mapsto_r (m', ref_a) \quad \mathtt{sel}(m', ref_a) = \bullet \quad rand() = (ref, n)}{(m, [a]) \mapsto_r (\mathtt{upd}(m', ref_a, (ref_a \mathrel{?} ref : n)), (ref_a \mathrel{?} ref : n))}$$

If v_a is a conditional value, then at least one of its definite forms is a reference, and in turn, v_a becomes definite to one of its definite references, as shown in the following rule:

$$\frac{(m, a) \mapsto_r (m', (ref' \mathrel{?} ref_a : n_a)) \quad \mathtt{sel}(\mathtt{def}(m', ref', 1), ref_a) = v \quad v \neq \bullet}{(m, [a]) \mapsto_r (\mathtt{def}(m', ref', 1), v)}$$

In the case that both definite forms are references, the interpreter has to make a random choice, which can easily described by non-deterministic rules. For instructions, the following rule describes the algorithm for interpreting move instructions:

$$\frac{(m, a) \mapsto_r (m', v_a) \quad (m, e) \mapsto_r (m'', v) \quad (m'', a) \mapsto_r (m'', ref) \quad Env_l(l_2) = i}{(m, l_1 : \ [a] \leftarrow e\ l_2) \mapsto_r (\mathtt{upd}(m'', ref, v), i)}$$

The address a of $[a]$ above is evaluated twice in order to make the order of interpretation irrelevant.

Again, due to limited space, we omit the proofs of soundness and completeness of the interpreter with respect to the *IL* semantics.

Escaping Memory Locations in SSA. In order to preserve SSA property we have to ensure that each memory word is written only once. In many compiler textbooks, memory is considered as a “variable”. We assume to have two new expressions to the *IL* syntax, one is `store` expression for creating a new value (of the entire memory), and the other is `load` which is similar to `sel` but occurs in the syntactic level.

Consider an excerpt of a program below and its SSA form:

$$\begin{array}{c} \mathtt{r}_1 \leftarrow [\mathtt{r}_2] \Rightarrow \mathtt{r}_1 \leftarrow \mathtt{load}(M_0, \mathtt{r}_2) \\ [\mathtt{r}_2] \leftarrow \mathtt{r}_1 \Rightarrow M_1 \leftarrow \mathtt{store}(M_0, \mathtt{r}_2, \mathtt{r}_1) \end{array}$$

This SSA form does not conform to the *IL* semantics. Recall that in the chunk memory model registers are part of the memory. Thus, the first instruction above updates memory M_0, but the second instruction uses the old M_0 as an argument of `store`.

An alternative solution to this problem is to leave the assignment $[a] \leftarrow e$ intact, but dynamically create and evaluate a new temporary t and an assignment $t \leftarrow e$ whenever the former assignment is evaluated. In detail, for every assignment $[a] \leftarrow e$, suppose a and e evaluate to ref and v respectively, we create an assignment of v to some chunk. The chunk is *associated* with ref and the point where $[a] \leftarrow e$ occurs.

First, we introduce a new environment Env_p which maps pairs of references and instruction labels (program points) to references. The resulting reference is said to be *associated* with the pair of reference and label given as arguments to Env_p. The following rule describes formally the above solution to the SSA problem:

$$\frac{(m, a) \mapsto ref \quad (m, e) \mapsto v \quad Env_l(l') = i' \quad Env_p(ref, l) = ref'}{m, l_1 : \; [a] \leftarrow e \; l_2 \mapsto (\mathtt{upd}(\mathtt{upd}(m, ref, v), ref', v), i')}$$

Note that, for the sake of clarity, the above rule assumes that we do not deal with any conditional value.

Introducing a new temporary for every escaping variable yields many histories to be analyzed. To reduce the number of histories, first we do not record values stored in memory locations of the current stack frame. These memory locations represent variables in the program. Since we dynamically create a new temporary every time we write to a stack memory location, the resulting temporaries are the SSA form of the variables represented by these memory locations. This condition also holds when a stack memory location represents more than one variable in the program.

Second, it is not necessary to create a new temporary if the escaping memory location is not represented by any stack memory location. That is, for any instruction $l_1 : \; [a] \leftarrow e \; l_2$ where a evaluates to (c, n) for some index n, but chunk c does not represent the stack, we do not create a new temporary. Again, for simplicity, the following definition assumes that we do not deal with any conditional value:

$$\frac{(m, a) \mapsto ref \quad ref = (\mathtt{c}_{sf}, n) \quad (m, e) \mapsto v \quad Env_l(l_2) = i' \quad Env_p(ref, l_1) = ref'}{m, l_1 : \; [a] \leftarrow e \; l_2 \mapsto (\mathtt{upd}(\mathtt{upd}(m, ref, v), ref', v), i')}$$

$$\frac{(m, a) \mapsto ref \quad ref = (c, n) \quad c \neq \mathtt{c}_{sf} \quad (m, e) \mapsto v \quad Env_l(l_2) = i'}{m, l_1 : \; [a] \leftarrow e \; l_2 \mapsto (\mathtt{upd}(m, ref, v), i')}$$

The chunk $\mathtt{c}_{sf}$ is the chunk representing the current stack frame.

Third, recall that writing to a memory location like (c, n) above gives rise to a program's side-effect. To be equal, the original program P and the optimized program P' must have the same side-effects. That is, for every instruction $l_1 : \; [a] \leftarrow e \; l_2$ in P there is a corresponding instruction $l'_1 : \; [a'] \leftarrow e' \; l'_2$ in P' such that evaluations of a and a' yield the same sequence of references, and also evaluations of e and e' yield the same sequence of values. For this purpose, we statically add a new instruction $l_0 : \; r \leftarrow a \; l_1$ before the instruction l_1, and a new instruction $l'_0 : \; r' \leftarrow a' \; l'_1$ before the instruction l'_1, where r and r' are new registers. Thus, in order to be equivalent, r and r' must correspond to each other. Moreover, having r and r', we do not have to create histories for the memory locations referred by a and a' above. Hence, the number of histories to be analyzed decreases.

The data produced by the randomized interpreter are then processed by an analyzer. The analyzer implements an algorithm for examining the value change sequences of all

variables, and guessing a basic block and variable correspondence. A full description of the analyzer is given in the extended version of this paper.

7 Experimental Results

This section describes the results of some experiments that have been conducted. The compiler used in the experiments is the *GNU C compiler* (*GCC*) 3.3.3. In every compilation, the compiler is instructed to dump the RTL, which is the intermediate representation used by the GCC, after performing the machine dependent reorganization. Then, the RTL dump is translated into the *IL* language, which in turn is interpreted by the randomized interpreter.

More precisely, in each experiment, programs are compiled twice, the first compilation is performed without any optimization (`O0`), and the second one with the (`O3`)-level optimization. The latter optimizations typically include *constant folding*, *copy* and *constant propagation*, *dead code* and *unreachable code elimination*, *algebraic simplification*, *local and global common subexpression elimination* followed by *jump optimization*, *partial redundancy elimination*, *loop invariant hoisting*, *induction variable elimination* and *strength reduction*, *branch optimization*, *loop inversion*, *loop reversal*, and *loop unrolling*.

For more extensive experiments we ran the randomized interpreter on the source code of GCC 3.3.3. The interpreter is developed incrementally, and the current implementation only supports 4-byte integer mode. The interpreter at the moment could interpret 299 functions out of $5,714$ functions which comprise the core of GCC. We are still developing the interpreter further to make it able to interpret all functions in the source of GCC. Most of the GCC functions that can be interpreted by the randomized interpreter are small functions; in average 25 lines of code.

Table 1 shows the result of running the interpreter on the source of GCC. We divide the table into several columns based on the size of the functions. Information that we obtain from the experiments is the number of points and variables in the correspondence, the number of visited variables during the interpretations, percentage of code coverage, visited branches, and time statistics. In the experiments the interpreter is set to execute at most $10,000$ lines of code.

For small functions whose sizes are less than 10 lines of code, the original and the optimized versions are almost the same. Table 1 shows that the interpreter could cover almost all code and visit all branches in these functions. Thus, the analyzer could produce a high percentage of block correspondence. For functions whose sizes are greater than 10 but less than 50 lines of code, the interpreter could still cover a large portion of the functions and also visit most of their branches. The code coverage is important since more lines of code that can be covered, the clearer the behaviors of the functions can be described, and the more block correspondence can be produced. Moreover, most of control variables are those used in the conditional expressions in the branches, so the more branches are visited the more point correspondence can be produced.

For functions, whose sizes are greater than 50 lines of code, the interpreter has problem with covering all code in these functions. Table 1 shows that more than 80% of code is not covered. This causes the percentage of point correspondence small. The code

Table 1. The result of running the randomized interpreter and the analyzer on the source of GCC

	$loc \leq 10$		$10 < loc \leq 25$		$25 < loc \leq 50$	
	P	P'	P	P'	P	P'
Blocks in correspondence	83.77%	81.92%	35.05%	29.56%	17.92%	20.43%
Variables in correspondence	91.97%	93.16%	76.55%	73.72%	87.90%	86.04%
Number of visited variables	12.28	10.46	26.2	18.61	19.33	20.4
Code coverage	90.16%	88.82%	50.37%	51.58%	30.71%	24.65%
Visited branches	97.74%	96.62%	72.66%	75.99%	41.21%	43.33%
Interpretation time	$0.056s$	$0.022s$	$0.170s$	$0.090s$%	$0.155s$	$0.046s$
Analysis time	$0.110s$	$0.102s$	$0.579s$	$0.617s$	$1.083s$	$0.653s$

	$50 < loc \leq 100$		$loc > 100$	
	P	P'	P	P'
Blocks in correspondence	11.24%	9.06%	8.06%	6.03%
Variables in correspondence	90.04%	67.01%	95.23%	67.08%
Number of visited variables	19.0	18.5	21.0	20.0
Code coverage	14.61%	15.81%	8.83%	10.67%
Visited branches	23.81%	25.25%	14.34%	17.34%
Interpretation time	$0.004s$	$0.002s$	$0.004s$	$0.001s$
Analysis time	$0.001s$	$0.002s$	$0.002s$	$0.001s$

coverage problem has long been known in program testing, that is to produce test cases that could cover all code in the function. We plan to tackle this problem by interpreting the function and its optimized version from points that are known to correspond to each other, and also combining our technique with symbolic interpretation to produce random inputs that can cover all code in these functions.

8 Discussion and Conclusion

Compared to the technique proposed in other papers, our technique is cheap since it requires no theorem proving or symbolic evaluation. Moreover, our technique can give reasonable results even in cases where other techniques do not work. For example, in the case of the loop reversal optimization our technique can still find correspondence between variables before and after the loop. But the price to pay is that the guessed correspondence has to be verified by a theorem prover. In most examples we studied such a verification was trivial. However, to get a full understanding of the technique, our system has to be combined with a VC generator and VC checker.

There are also examples when our technique may not be appropriate. For example, using random values may be inappropriate when one of the branches is “hard” to reach. Gulwani and Necula [3] propose a very interesting technique, also based on random inputs, for solving this problem, but it is only applicable to a very special class of programs. We believe that our technique can be improved both by “correcting” random values as in [3] and also by mixing symbolic interpretation with the random one.

The definition of basic block and variable correspondences in Section 3 does not capture some optimizing transformations. For example, consider the following programs:

P :
x_1 := 0;
do
  i := $i + 1$;
  x_1 := 1;
  x_2 := $i + x_1$;
while $i < n$
x_3 := x_1;

P' :
x_1 := 1;
do
  i := $i + 1$;
  x_2 := $i + x_1$;
while $i < n$
x_3 := x_1;

P' is obtained by applying loop invariant hoisting and dead code elimination to P. The loop bodies of the two programs correspond to each other. Moreover, the loop body of P also corresponds to the assignment x_1 := 1 in P' since an instance of this assignment is executed many times in P. However, our definition of correspondence does not capture this case. Indeed, it is hard to give a simple but general formal definition of basic block and variable correspondence that can capture all existing optimizing transformations. Our definition can be extended further to capture more transformations. For example, by adding some properties into the definition to allow a block to correspond to more than one other block in the runs will capture the correspondence of the above programs. Although our definition of correspondence does not capture the above case, in the SSA form, the randomized interpreter and the analyzer can produce the correspondence of the loop body of P and the assignment x_1 := 1 in P'.

We are still improving the definition of correspondence. The definition we provide in this paper is considerably simple and easy-to-understand, but nonetheless captures the notion of correspondence needed to prove the equivalence of two programs. Particularly in the above example, without the correspondence of x_1, but as long as we can establish the correspondence of variables i, n, x_2, and x_3, the equivalence of P and P' can easily be proved. Indeed, the randomized interpreter and the analyzer can establish such a correspondence.

References

1. B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages (POPL 1988)*, pages 1–11, 1988.
2. Yi Fang. Personal communication over emails, 2005.
3. S. Gulwani and G.C. Necula. Global value numbering using random interpretation. In N.D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 342–352. ACM Press, 2004.
4. Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. Capturing the effects of code improving transformations. In *IEEE PACT*, pages 118–123, 1998.
5. Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

6. George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI)*, pages 83–95, June 2000.
7. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *LNCS*, 1384, 1998.
8. Amir Pnueli, Lenore Zuck, Yi Fang, Benjamin Goldberg, and Ying Hu. Translation and run-time validation of optimized code. *ENTCS*, 70(4):1–22, 2002.
9. Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13. ACM Press, 2004.
10. Robert van Engelen, David B. Whalley, and Xin Yuan. Automatic validation of code-improving transformations. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 206–210. Springer-Verlag, 2001.
11. L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. Voc: A methodology for the translation validation of optimizing compilers. *j-jucs*, 9(3):223–247, March 2003.

Boolean Heaps

Andreas Podelski and Thomas Wies

Max-Planck-Institut für Informatik, Saarbrücken, Germany
{podelski, wies}@mpi-inf.mpg.de

Abstract. We show that the idea of predicates on heap objects can be cast in the framework of predicate abstraction. This leads to an alternative view on the underlying concepts of three-valued shape analysis by Sagiv, Reps and Wilhelm. Our construction of the abstract post operator is analogous to the corresponding construction for classical predicate abstraction, except that predicates over objects on the heap take the place of state predicates, and boolean heaps (sets of bitvectors) take the place of boolean states (bitvectors). A program is abstracted to a program over boolean heaps. For each command of the program, the corresponding abstract command is effectively constructed by deductive reasoning, namely by the application of the *weakest precondition* operator and an entailment test. We thus obtain a symbolic framework for shape analysis.

1 Introduction

The transition graph of a program is formed by its states and the transitions between them. The idea of *predicate abstraction* [6] (used in a tool such as SLAM [2]) is to abstract a state by its evaluation under a number of given state predicates; each edge between two concrete states in the transition graph gives rise to an edge between the two corresponding abstract states. One thus abstracts the transition graph to a graph over abstract states.

For a program manipulating pointers, each state is represented by a *heap graph*. A heap graph is formed by the allocated objects in the heap and pointer links between them. The idea of *three-valued shape analysis* [13] is to apply to the heap graph the same abstraction that we have applied to the transition graph. One abstracts an object in the heap by its evaluation under a number of *heap predicates*; edges between concrete objects in the heap graph give rise to edges between the corresponding abstract objects. One thus abstracts a heap graph to a graph over abstract objects.

The analogy between predicate abstraction and the abstraction proposed in three-valued shape analysis is remarkable. It does not seem helpful, however, when it comes to the major challenge: how can one compute the abstraction of the transition graph when states are heap graphs and the abstraction is defined on objects of the heap graph? This paper answers a refinement of this question, namely whether the abstraction can be defined and computed in the formal setup and with the basic machinery of predicate abstraction.

Our technical contributions that are needed to accomplish this task are summarized as follows:

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 268–283, 2005.
© Springer-Verlag Berlin Heidelberg 2005

- We omit explicit edges between abstract objects in the abstract state, since we observe that one can also encode edge relations implicitly using appropriate heap predicates on objects. This makes it possible to define the abstract post operator by *local* updates of the values of heap predicates.
- We show that one can implement the abstraction by a simple source-to-source transformation of a pointer program to an abstract finite-state program which we call a *boolean heap program*. This transformation is analogous to the corresponding transformation in predicate abstraction, except that predicates over objects on the heap take the place of state predicates and boolean heaps (sets of bitvectors) take the place of boolean states (bitvectors).
- We formally identify the post operator of a boolean heap program as an abstraction of the best abstract post operator on an abstract domain of formulas. For each command of the program, the corresponding abstract command is constructed by the application of a *weakest precondition* operator on heap predicates and an entailment test (implemented by a syntactic substitution resp. by a call to a theorem prover).

Outline. In Section 2 we give related work; in particular, we summarize the key concepts of predicate abstraction. Section 3 gives the algorithmic description of our analysis. Section 4 defines the formal semantics of programs manipulating pointers. In Section 5 we give a theory of heap predicates that extends the notion of state predicates and state predicate transformers to predicates on heap objects and heap predicate transformers. Section 6 provides a formal definition of our analysis in the framework of abstract interpretation. In Section 7 we formally identify the abstract system described in Section 3 as a composition of additional abstraction functions with the best abstract post operator on our abstract domain. Section 8 concludes. Omitted proofs can be found in the extended version of this paper[1].

2 Related Work

In [13] Sagiv, Reps and Wilhelm describe a parametric framework to shape analysis based on three-valued logic. They abstract sets of states by three-valued logical structures. The abstraction is defined in terms of equivalence classes of objects in the heap that are induced by a finite set of predicates on heap objects. We use several ideas from this approach. In particular, there is a strong connection between their abstract domain and ours: a translation from three-valued logical structures, as they arise in [13], into formulas in first-order logic is given in [15]. Shape analysis constraints [10] extend this translation to a boolean algebra of state predicates that is isomorphic to the class of three-valued logical structures in [13]; our abstract domain is a fragment of shape analysis constraints.

[1] Available on the web at `http://www.mpi-inf.mpg.de/~wies/papers/boolean-heaps-extended.pdf`

In [16] a symbolic algorithm is presented that can be used for shape analysis *à la* [13]. It is based on an *assume* operation that is implemented using a decision procedure. The *assume* operation allows inter-procedural shape analysis based on assume-guarantee reasoning. Moreover, *assume* can be instantiated to compute best abstraction functions, most-precise post operators, and the meet operation for abstract domains of three-valued logical structures. In our framework we do not depend on an intermediate representation of sets of states in terms of three-valued logical structures. We work exclusively on formulas.

PALE [12] is a Hoare-style system for the analysis of pointer programs that is based on weak monadic second order logic over trees. Its degree of automation is restricted, because loops in the program have to be manually annotated with loop invariants. Also the class of data structures that PALE is able to handle is restricted to graph types [9]. In our approach we synthesize loop invariants automatically. Furthermore, our analysis is not restricted *a priori* to a particular class of data structures; which data structures our analysis is able to treat only depends on the capabilities of the underlying theorem prover that is used to compute the abstraction.

Software model checkers such as SLAM [2] use predicate abstraction [6] to abstract the concrete transition system into a finite-state boolean program. A state of the resulting boolean program, i.e. an abstract state, is given by a bitvector over the abstraction predicates. Each transition of the concrete system gives rise to a corresponding simultaneous update of the predicate values in the boolean program.

General scheme	**Example**
Concrete command:	Concrete command:
c	**var** x : integer
	$x := x + 1$
State predicates:	State predicates:
$\mathcal{P} = \{p_1, \ldots, p_n\}$	$p_1 \stackrel{def}{=} x = 0, \qquad p_2 \stackrel{def}{=} x > 0$
Abstract boolean program:	Abstract boolean program:
var $p_1, \ldots, p_n$: boolean	**var** p_1, p_2 : boolean
for each $p_i \in \mathcal{P}$ **do**	**if** false **then** $p_1 :=$ true
if $\mathsf{wp}^{\#}\, c\, p_i$ **then** $p_i :=$ true	**else if** $p_1 \vee p_2$ **then** $p_1 :=$ false
else if $\mathsf{wp}^{\#}\, c\, (\neg p_i)$ **then** $p_i :=$ false	**else** $p_1 := *$
else $p_i := *$	**if** $p_1 \vee p_2$ **then** $p_2 :=$ true
	else if $\neg p_1 \wedge \neg p_2$ **then** $p_2 :=$ false
	else $p_2 := *$

Fig. 1. Construction of a boolean program from a concrete command via predicate abstraction. All predicates are updated simultaneously. The value '*' stands for non-deterministic choice.

Figure 1 shows the transformation of a concrete command to the corresponding predicate updates in the abstract boolean program. The actual abstraction step lies in the computation of $\mathsf{wp}^{\#}\ c\ p$ – the best boolean under-approximation (in terms of abstraction predicates) of the weakest precondition of predicate p and command c (for example false is the best under-approximation of $\mathsf{wp}^{\#}(x := x + 1)\ (x = 0)$ with respect to p_1 and p_2). One of the advantages of predicate abstraction is that the computation of this operator can be done *offline* in a pre-processing step (using a decision procedure or theorem prover). Therefore, one has a clear separation between the abstraction phase and the actual fixed point computation of the analysis.

There are several approaches that use classical predicate abstraction for shape analysis; see e.g. [5] and [1]. As discussed in [11], if one wants to gain the same precision with classical predicate abstraction as for the abstract domain proposed in [13] then in general one needs an exponential number of state predicates compared to the number of predicates on heap objects that are used in [13]. This seems to be the major drawback of using standard predicate abstraction for shape analysis. We solve this problem by combining the core ideas from both frameworks. In particular, we use Cartesian abstraction in a way that is reminiscent of the approach described in [3]. However, we restrict our attention to safety properties, whereas in [1] also liveness properties are considered.

3 Boolean Heap Programs

Our analysis proceeds as follows: (1) we choose a set of predicates over heap objects for the abstraction (defining the abstract domain); (2) we construct an abstract finite-state program in analogy to predicate abstraction (the abstract post operator); and (3) we apply finite-state model checking to the abstract program (the fixed point computation). In the following we explain in detail how the abstract domain and the construction of the abstract program look like.

For an abstract domain given by graphs over abstract objects it is difficult to compute the abstract post operator as an operation on the whole abstract state. Instead one would like to represent the abstract post operator corresponding to a pointer command by *local* updates. Local means that one updates each abstract object in isolation. However, pointer commands update pointer fields. The problem is: how can one account for the update of pointer fields by local updates on abstract objects?

The key idea is that we use a set of abstract objects to represent an abstract state, i.e. we omit edges between abstract objects. A state s is represented by a set of abstract objects, if every concrete object in s is represented by one abstract object in the set. Instead of having explicitly-encoded pointer relations in the abstract state, pointer information is implicitly encoded using appropriate predicates on heap objects for the abstraction. In particular, the presence or absence of an edge between two abstract objects can be encoded into heap predicates on objects. Adding these predicates to the set of abstraction predicates will preserve this information in the abstraction; see [14].

General scheme

Concrete command:

c

Unary heap predicates:

$\mathcal{P} = \{p_1(v), \ldots, p_n(v)\}$

Boolean heap program:

var V : set of bitvectors over $\mathcal{P}$
for each $\overline{p} \in V$ **do**
  for each $p_i \in \mathcal{P}$ **do**
    if $\overline{p} \models \mathsf{hwp}^{\#}\ c\ p_i$
      then $\overline{p}.p_i := \mathsf{true}$
    else if $\overline{p} \models \mathsf{hwp}^{\#}\ c\ (\neg p_i)$
      then $\overline{p}.p_i := \mathsf{false}$
    else $\overline{p}.p_i := *$

Example

Concrete command:

var x, y, z : list
$x.next := y$

Unary heap predicates:

$p_1(v) \stackrel{def}{=} x = v, \quad p_2(v) \stackrel{def}{=} y = z$
$p_3(v) \stackrel{def}{=} next(v) = z$

Boolean heap program:

var V : set of bitvectors over $\{p_1, p_2, p_3\}$
for each $\overline{p} \in V$ **do**
  if $\overline{p} \models p_1$ **then** $\overline{p}.p_1 := \mathsf{true}$
  else if $\overline{p} \models \neg p_1$ **then** $\overline{p}.p_1 := \mathsf{false}$
  if $\overline{p} \models p_2$ **then** $\overline{p}.p_2 := \mathsf{true}$
  else if $\overline{p} \models \neg p_2$ **then** $\overline{p}.p_2 := \mathsf{false}$
  if $\overline{p} \models \neg p_1 \wedge p_3 \vee p_1 \wedge p_2$ **then**
    $\overline{p}.p_3 := \mathsf{true}$
  else if $\overline{p} \models \neg(\neg p_1 \wedge p_3 \vee p_1 \wedge p_2)$
    then $\overline{p}.p_3 := \mathsf{false}$

Fig. 2. Transformation of a concrete command into a boolean heap program

The set of abstract objects defining the abstract state is represented by a set of bitvectors over abstraction predicates; we call such a set of bitvectors a boolean heap. We abstract a pointer program by a *boolean heap program* as defined in Fig. 2. The construction naturally extends the one used in predicate abstraction which is given in Fig. 1. The difference is that a state of the abstract program is not given by a single bitvector, but by a set of bitvectors, i.e a boolean heap. Transitions in boolean heap programs change the abstract state via local updates on abstract objects ($\overline{p}.p_i := \mathsf{true}$) rather than global updates on the whole abstract state ($p_i := \mathsf{true}$). Consequently, we replace the abstraction of the weakest precondition operator on state predicates $\mathsf{wp}^{\#}$ by the abstraction of a weakest precondition operator on heap predicates $\mathsf{hwp}^{\#}$. While causing only a moderate loss of precision, this construction avoids the exponential blowup in the construction of the abstract program that occurs when standard predicate abstraction is used to simulate a graph based abstract domain with an appropriate set of state predicates.

In the rest of the paper we give a formal account of boolean heap programs. In particular, we make precise what it means to compute the operator $\mathsf{hwp}^{\#}$. Furthermore, we identify the post operator that corresponds to a boolean heap program as an abstraction of the best abstract post operator on boolean heaps. This way we identify the points in the analysis where we can lose precision.

4 Pointer Programs

We consider the language of pointer programs defined in Fig. 3. In order to highlight our main observations, we make several simplifications: (1) we do not model the program counter; (2) we do not consider allocation or deallocation of objects; and (3) we do not treat null pointers explicitly; in particular, we do not treat dereferences of null pointers. However, none of these simplifications imposes inherent restrictions of our results.

A state of the program is represented as a logical structure over the vocabulary of program variables *Var* and pointer fields *Field*. Since we do not treat allocation and deallocation of objects, we fix a set of objects *Heap* that is not changed during program execution and serves as the common universe of all program states. Therefore, a state degenerates to an interpretation function, i.e. a valuation of program variables to elements of *Heap* and pointer fields to total functions over *Heap*. Note that we define states as a Cartesian product of interpretation functions, but for notational convenience we implicitly project to the appropriate component function when a symbol is interpreted in a state.

The transition relation of a pointer program gives rise to the definition of the standard predicate transformers. The predicate transformers post (strongest postcondition) and wp (weakest precondition) are defined as usual.

5 Heap Predicates

We will abstractly represent sets of program states using formulas. We consider a logic given with respect to the signature of program variables *Var* and pointer fields *Field*. Terms in formulas are built from constant symbols $x \in Var$ that are interpreted as heap objects and function symbols $f \in Field$ that are interpreted as functions on heap objects. Formulas are interpreted in states (together with a variable assignment for the free variables). The following discussion is not restricted to a particular logic. The only further assumption we make is that the logic is closed under syntactic substitutions.

Formulas may contain free first-order variables $v_1, \ldots, v_n$. There are two equivalent ways to define the denotation of such formulas. As a running example consider the formula $\varphi(v)$ with free variable v given by:

$$\varphi(v) \equiv \; f(v) = z \;\; .$$

The intuitive way of defining the denotation $[\![\varphi(v)]\!]$ of $\varphi(v)$ is a function mapping a state s to the set of heap objects that when assigned to the free variable v satisfy φ in s:

$$\lambda\, s \in State\,.\, \{\, o \in Heap \mid s\ f\ o = s\ z \,\} \;\; .$$

For technical reasons we use an equivalent definition. Namely, we define $[\![\varphi(v)]\!]$ as a function mapping an object o to the set of all states in which φ holds if v is assigned to o:

$$[\![\varphi(v)]\!] = \lambda\, o \in Heap\,.\, \{\, s \in State \mid s\ f\ o = s\ z \,\} \;\; .$$

Syntax of expressions and commands:

$$x \in \mathit{Var} \; - \; \text{set of program variables}$$
$$f \in \mathit{Field} \; - \; \text{set of pointer fields}$$

$$e \in \mathit{OExp} ::= x \mid e.f$$
$$b \in \mathit{BExp} ::= e_1 = e_2 \mid \neg b \mid b_1 \wedge b_2$$
$$c \in \mathit{Com} ::= e_1 := e_2 \mid \mathsf{assume}(b)$$

Semantics of expression and commands:

$$o \in \mathit{Heap} \; - \; \text{nonempty set of allocated objects}$$
$$s \in \mathit{State} \stackrel{def}{=} (\mathit{Var} \to \mathit{Heap}) \times (\mathit{Field} \to \mathit{Heap} \to \mathit{Heap})$$

$$[\![x]\!] \; s \stackrel{def}{=} s \; x \qquad [\![e_1 = e_2]\!] \; s \stackrel{def}{=} [\![e_1]\!] \; s = [\![e_2]\!] \; s$$
$$[\![e.f]\!] \; s \stackrel{def}{=} s \; f \; ([\![e]\!] \; s) \qquad [\![\neg b]\!] \; s \stackrel{def}{=} \neg([\![b]\!] \; s)$$
$$[\![b_1 \wedge b_2]\!] \; s \stackrel{def}{=} [\![b_1]\!] \; s \wedge [\![b_2]\!] \; s$$

$$[\![x := e]\!] \; s \; s' \stackrel{def}{=} s' = s[x \mapsto [\![e]\!] \; s]$$
$$[\![e_1.f := e_2]\!] \; s \; s' \stackrel{def}{=} s' = s[f \mapsto (s \; f)[[\![e_1]\!] \; s \mapsto [\![e_2]\!] \; s]]$$
$$[\![\mathsf{assume}(b)]\!] \; s \; s' \stackrel{def}{=} [\![b]\!] \; s \wedge s = s'$$

Predicate transformers:

$$\mathsf{post}, \mathsf{wp} \in \mathit{Com} \to 2^{\mathit{State}} \to 2^{\mathit{State}}$$
$$\mathsf{post} \; c \; S \stackrel{def}{=} \{ s' \mid \exists s. [\![c]\!] \; s \; s' \wedge s \in S \}$$
$$\mathsf{wp} \; c \; S \stackrel{def}{=} \{ s \mid \forall s'. [\![c]\!] \; s \; s' \Rightarrow s' \in S \}$$

Fig. 3. Syntax and semantics of pointer programs

Definition 1 (Heap Formulas and Heap Predicates). *Let $\varphi(\overline{v})$ be a formula with n free first-order variables $\overline{v} = (v_1, \ldots, v_n)$. The denotation $[\![\varphi(\overline{v})]\!]$ of $\varphi(\overline{v})$ is defined by:*

$$[\![\varphi(\overline{v})]\!] \stackrel{def}{=} \lambda \overline{o} \in \mathit{Heap}^n . \{ s \in \mathit{State} \mid s, [\overline{v} \mapsto \overline{o}] \models \varphi(\overline{v}) \} \; .$$

We call the denotation $[\![\varphi(v)]\!]$ an n-ary heap predicate*. The set of all heap predicates is given by:*

$$\mathit{HeapPred}[n] \stackrel{def}{=} \mathit{Heap}^n \to 2^{\mathit{State}} .$$

We skip the parameter n for heap predicates whenever this causes no confusion. Moreover, we consider 0-ary heap predicates as state predicates and call closed heap formulas *state formulas*.

We want to implement predicate transformers (which are operations on sets of states) through operations on heap formulas. However, heap formulas denote heap predicates rather than sets of states. We now exploit the fact that for a heap predicate p we have that $p\,\overline{o}$ is a set of states. This allows us to generalize the predicate transformers from sets of states to heap predicates.

Definition 2 (Heap Predicate Transformers). *The predicate transformers* post *and* wp *are lifted to* heap predicate transformers *as follows:*

$$\begin{aligned} \mathsf{hpost}, \mathsf{hwp} &\in Com \rightarrow HeapPred \rightarrow HeapPred \\ \mathsf{hpost}\ c\, p &\stackrel{def}{=} \lambda\, \overline{o}\,.\ \mathsf{post}\ c\ (p\, \overline{o}) \\ \mathsf{hwp}\ c\, p &\stackrel{def}{=} \lambda\, \overline{o}\,.\ \mathsf{wp}\ c\ (p\, \overline{o})\ . \end{aligned}$$

Since the heap predicate transformers are obtained from the standard predicate transformers via a simple lifting, their characteristic properties are preserved. In particular, the following proposition holds.

Proposition 1. *Let c be a command. The heap predicate transformers* hpost *and* hwp *form a Galois connection on the boolean algebra of heap predicates, i.e. for all $p, p' \in HeapPred[n]$ and $\overline{o} \in Heap^n$:*

$$\mathsf{hpost}\ c\, p\, \overline{o} \subseteq p'\, \overline{o} \iff p\, \overline{o} \subseteq \mathsf{hwp}\ c\, p'\, \overline{o}\ .$$

The operator hwp is one of the ingredients that we need to construct boolean heap programs. Therefore, it is important that it can be characterized in terms of a syntactic operation on formulas. Ideally this operation does not introduce additional quantifiers. Such a characterization of hwp exists, because the transition relation is deterministic. For the command $c = (x.f := y)$ we have e.g.:

$$\begin{aligned} \mathsf{hwp}\ c\, [\![\varphi(v)]\!] &= \lambda\, o\,.\ \{\, s \mid \forall s'.\, [\![c]\!]\ s\ s' \Rightarrow s' \in ([\![\varphi(v)]\!]\ o)\,\} \\ &= [\![\varphi(v)[f := \lambda\, v\,.\ \text{if } v = x \text{ then } y \text{ else } f(v)]]\!] \\ &= [\![v = x \wedge y = z\ \vee\ v \neq x \wedge f(v) = z]\!]\ . \end{aligned}$$

The resulting formula denotes the object in a state s whose f-successor is pointed to by z in the successor state of s under c. The correctness of the above transformation is justified by the following proposition.

Proposition 2. *Let $\varphi(\overline{v})$ be a heap formula. The operator* hwp *applied to the denotation of $\varphi(\overline{v})$ reduces to a syntactic operation:*

$$\begin{aligned} \mathsf{hwp}\ (x := e)\ [\![\varphi(\overline{v})]\!] &= [\![\varphi(\overline{v})[x := e]]\!] \\ \mathsf{hwp}\ (e_1.f := e_2)\ [\![\varphi(\overline{v})]\!] &= [\![\varphi(\overline{v})[f := \lambda\, v\,.\ \text{if } v = e_1 \text{ then } e_2 \text{ else } f(v)]]\!] \\ \mathsf{hwp}\ (\mathsf{assume}(b))\ [\![\varphi(\overline{v})]\!] &= [\![b\ \rightarrow\ \varphi(\overline{v})]\!]\ . \end{aligned}$$

Note that the lambda terms do not cause any problems even if we restrict to first-order logics. The function symbols that are substituted by lambda terms

always occur in β-redexes, i.e. as in the example above, it is always possible to rewrite the result of the substitution to an equivalent lambda-free formula.

Due to Prop. 2 it is convenient to overload hwp both to a function on heap predicates as well as a function on heap formulas. Whenever we apply hwp to a heap formula we refer to the corresponding syntactic operation given in Prop. 2.

6 Heap Predicate Abstraction

We systematically construct an abstract post operator by following the framework of abstract interpretation [4]. Hence, we need to provide an abstract domain, as well as an abstraction and meaning function.

We propose an abstract domain that is given by a set of state formulas and is parameterized by unary heap predicates. For the rest of the paper we fix a particular finite set of unary heap predicates $\mathcal{P}$. We consider $\mathcal{P}$ to be given as a set of heap formulas with one dedicated free variable v. For notational convenience we consider $\mathcal{P}$ to be closed under negation.

Definition 3 (Boolean Heaps). *A* boolean heap *over $\mathcal{P}$ is a formula Ψ of the form:*

$$\Psi = \forall v. \bigvee_i C_i(v)$$

where each $C_i(v)$ is a conjunction of heap predicates in $\mathcal{P}$. We denote the set of all boolean heaps over $\mathcal{P}$ by BoolHeap.

In order to allow our analysis to treat joins in the control flow adequately, we take the disjunctive completion over boolean heaps as our abstract domain.

Definition 4 (Abstract Domain). *The* abstract domain *over $\mathcal{P}$ is the pair $\langle AbsDom, \models \rangle$, where AbsDom is given by all disjunctions of boolean heaps:*

$$AbsDom \stackrel{def}{=} \{ \bigvee_{\Psi \in F} \Psi \mid F \subseteq_{fin} BoolHeap \} \ .$$

The partial order $\models$ on elements in AbsDom is the entailment relation on formulas.

A boolean heap can be represented as a set of bitvectors over $\mathcal{P}$, one bitvector for each conjunction. Hence, it is easy to see that the abstract domain is isomorphic to sets of sets of bitvectors over $\mathcal{P}$. Moreover, the abstract domain is finite and both closed under disjunctions and conjunctions[2]. Therefore, it forms a complete lattice.

The meaning function γ that maps elements of the abstract domain to sets of states is naturally given by the denotation function, i.e. each formula Ψ of the abstract domain is mapped to the set of its models:

$$\gamma \in AbsDom \to 2^{State}$$
$$\gamma\, \Psi \stackrel{def}{=} [\![\Psi]\!] \ .$$

[2] Conjunctions distribute over the universal quantifiers in boolean heaps.

The abstraction function α is determined by:

$$\alpha \in 2^{State} \to AbsDom$$
$$\alpha\ S \stackrel{def}{=} \bigwedge \{\, \Psi \in AbsDom \mid S \subseteq [\![\Psi]\!] \,\} \ .$$

The function γ distributes over disjunctions and is thus a complete meet morphism. Together with the fact that we defined α as the best abstraction function with respect to γ, we can conclude that α and γ form a Galois connection between concrete and abstract domain:

$$\langle 2^{State}, \subseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle AbsDom, \models \rangle \ .$$

If we consider a state s, the abstraction function α maps the singleton $\{s\}$ to the smallest boolean heap that is valid in s. This boolean heap describes the boolean covering of heap objects with respect to the heap predicates $\mathcal{P}$.

In order to describe these smallest boolean coverings, we assign an *abstract object* $\alpha_s(o)$ to every object o and state s. This abstract object is given by a monomial (complete conjunction) of heap predicates and represents the equivalence class of all objects that satisfy the same heap predicates as o in s:

$$\alpha_s(o) \stackrel{def}{=} \bigwedge \{\, p(v) \in \mathcal{P} \mid s, [v \mapsto o] \models p(v) \,\} \ .$$

The smallest boolean heap that abstracts s consists of all abstract objects $\alpha_s(o)$ for objects $o \in Heap$. Formally, the abstraction of a set of states S is characterized as follows:

$$\alpha\ S \equiv \bigvee_{s \in S} \forall v. \bigvee_{o \in Heap} \alpha_s(o) \ .$$

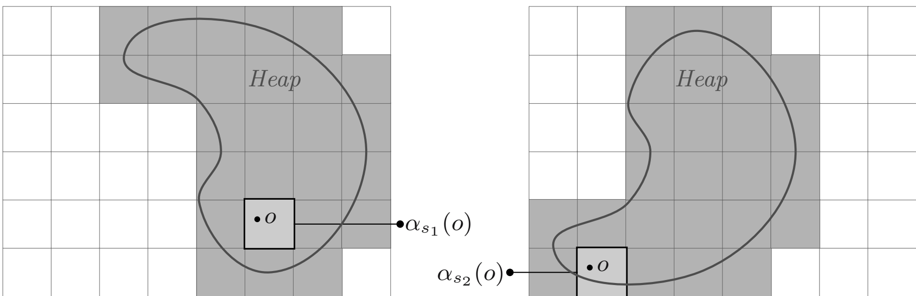


Fig. 4. The boolean heaps for two states s_1 and s_2. The same object $o \in Heap$ falls into different equivalence classes $\alpha_{s_1}(o)$ and $\alpha_{s_2}(o)$ for each of the states s_1 and s_2. This leads to a different boolean covering of the set *Heap* in the two states and hence to different boolean heaps.

7 Cartesian Abstraction

According to [4] the best abstract post operator $\mathsf{post}^{\#}$ is given by the composition of α, post and γ. In the following, we fix a command c and consider all applications of predicate transformers with respect to this particular command. Using the characterization of α from the previous section, we get:

$$\mathsf{post}^{\#}(\Psi) \stackrel{def}{=} \alpha \circ \mathsf{post} \circ \gamma(\Psi) \equiv \bigvee_{s \in [\![\Psi]\!]} \forall v. \bigvee_{o \in Heap} \alpha_{\mathsf{post}(\{s\})}(o) \ .$$

In order to compute the image of Ψ under $\mathsf{post}^{\#}$ we need to check for each boolean heap whether it appears as one of the disjuncts in $\mathsf{post}^{\#}(\Psi)$. Given that n is the number of (positive) heap predicates in $\mathcal{P}$, considering all 2^{2^n} boolean heaps explicitly results in a doubly-exponential running time for the computation of $\mathsf{post}^{\#}$. Therefore, our goal is to develop an approximation of the best abstract post operator that can be easily implemented. However, we require this operator to be formally characterized in terms of an abstraction of $\mathsf{post}^{\#}$.

Since the best abstract post operator distributes over disjunctions, we characterize the abstraction of $\mathsf{post}^{\#}$ on boolean heaps rather than their disjunctions. In the following, consider the boolean heap Ψ given by:

$$\Psi = \forall v. \psi(v) \ .$$

As illustrated in Fig. 5, the problem is that even if we apply $\mathsf{post}^{\#}$ to the single boolean heap Ψ, its image under $\mathsf{post}^{\#}$ will in general be a disjunction of boolean heaps. We first abstract a disjunction of boolean heaps by a single boolean heap. This is accomplished by merging all coverings represented by boolean heaps in $\mathsf{post}^{\#}(\Psi)$ into a single one. That means the resulting single boolean heap represents a covering of all objects for all states that are models of $\mathsf{post}^{\#}(\Psi)$.

We define the best abstractions of the heap predicate transformers with respect to the set of all boolean combinations of heap predicates in $\mathcal{P}$ (denoted by $\mathcal{BC}(\mathcal{P})$):

$$\mathsf{hpost}^{\#}(\psi(v)) \stackrel{def}{=} \bigwedge \{ \varphi(v) \in \mathcal{BC}(\mathcal{P}) \mid \psi(v) \models \mathsf{hwp}(\varphi(v)) \}$$
$$\mathsf{hwp}^{\#}(\psi(v)) \stackrel{def}{=} \bigvee \{ \varphi(v) \in \mathcal{BC}(\mathcal{P}) \mid \varphi(v) \models \mathsf{hwp}(\psi(v)) \} \ .$$

By Prop. 1 and the definition of $\mathsf{hpost}^{\#}$ respectively $\mathsf{hwp}^{\#}$ it is easy to see that these two operators again form a Galois connection on the set of all boolean combinations of heap predicates in $\mathcal{P}$.

Formally, the approximation of the best abstract post that we described above corresponds to the application of the best abstraction of the operator hpost to the heap formula $\psi(v)$.

Proposition 3. *Let $\Psi = \forall v. \psi(v)$ be a boolean heap. Applying* $\mathsf{hpost}^{\#}$ *to $\psi(v)$ results in an abstraction of* $\mathsf{post}^{\#}(\Psi)$*:*

$$\mathsf{post}^{\#}(\Psi) \models \forall v. \, \mathsf{hpost}^{\#}(\psi(v)) \ .$$

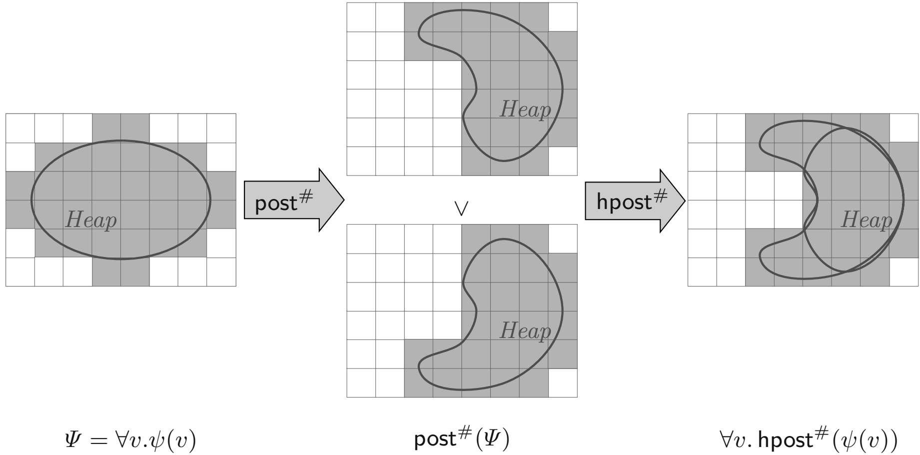


Fig. 5. Application of $\mathsf{post}^\#$ to a single boolean heap Ψ and its approximation using $\mathsf{hpost}^\#$

Since the operator $\mathsf{hpost}^\#$ again distributes over disjunctions, we can compute the new covering by applying $\mathsf{hpost}^\#$ *locally* to each disjunct in $\psi(v)$. That is, if $\psi(v)$ is given as a disjunction of abstract objects:

$$\psi(v) = \bigvee_i C_i(v)$$

then for each $C_i(v)$ we compute the new covering $\mathsf{hpost}^\#(C_i(v))$ of objects represented by $C_i(v)$ for the states satisfying $\mathsf{post}^\#(\Psi)$.

However, computing this localized post operator is still an expensive operation. The result of $\mathsf{hpost}^\#$ applied to an abstract object will in general be a disjunction of abstract objects. We face the same problem as before: we would have to consider all 2^n monomials over heap predicates, in order to compute the precise image of a single abstract object under $\mathsf{hpost}^\#$.

A disjunction of conjunctions over abstraction predicates can be represented as a set of bitvectors. In the context of predicate abstraction one uses Cartesian abstraction to approximate sets of bitvectors [3]. For a set of bitvectors represented by a boolean formula $\psi(v)$, the Cartesian abstraction $\alpha_{\mathsf{Cart}}(\psi(v))$ is given by the smallest conjunction over abstraction predicates that is implied by $\psi(v)$:

$$\alpha_{\mathsf{Cart}}(\psi(v)) \stackrel{def}{=} \bigwedge \{\, p(v) \in \mathcal{P} \mid \psi(v) \models p(v) \,\} \ .$$

Figure 6 sketches the idea of Cartesian abstraction in our context. It abstracts all abstract objects in the image under the operator $\mathsf{hpost}^\#$ by a single conjunction. Composing the operator $\mathsf{hpost}^\#$ with the Cartesian abstraction function gives us our final abstraction of the best abstract post operator.

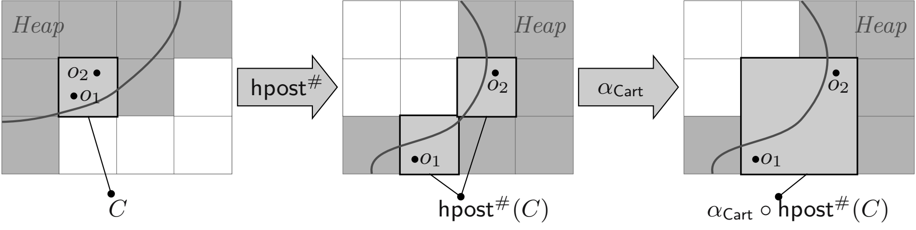


Fig. 6. Application of $\mathsf{hpost}^\#$ to a single abstract object C and the approximation under α_{Cart}

Definition 5 (Cartesian Post). *Let* $\Psi = \forall v. \bigvee_i C_i(v)$ *be a boolean heap. The* Cartesian post *of* Ψ *is defined as follows:*

$$\mathsf{post}^\#_{\mathsf{Cart}}(\Psi) \stackrel{def}{=} \forall v. \bigvee_i \alpha_{\mathsf{Cart}} \circ \mathsf{hpost}^\#(C_i(v)) \ .$$

We extend the Cartesian post to a function on AbsDom in the natural way by pushing it over disjunctions of boolean heaps.

Theorem 1 (Soundness of Cartesian Post). *The Cartesian post is an abstraction of* $\mathsf{post}^\#$*:*

$$\forall \Psi \in \mathit{BoolHeap}.\ \mathsf{post}^\#(\Psi) \models \mathsf{post}^\#_{\mathsf{Cart}}(\Psi) \ .$$

Proof. Let $\Psi = \forall v. \bigvee_i C_i(v)$ be a boolean heap. The statement follows immediately from Prop. 3 and the fact that for every $C_i(v)$ we have $C_i(v) \models \alpha_{\mathsf{Cart}}(C_i(v))$.

Theorem 2 (Characterization of Cartesian Post). *Let* $\Psi = \forall v. \bigvee_i C_i(v)$ *be a boolean heap. The Cartesian post of* Ψ *is characterized as follows:*

$$\mathsf{post}^\#_{\mathsf{Cart}}(\Psi) \equiv \forall v. \bigvee_i \bigwedge \{\, p(v) \in \mathcal{P} \mid C_i(v) \models \mathsf{hwp}^\#(p(v)) \,\} \ .$$

Proof. Using the fact that $\mathsf{hpost}^\#$ and $\mathsf{hwp}^\#$ form a Galois connection $(*)$ we have:

$$\begin{aligned} \mathsf{post}^\#_{\mathsf{Cart}}(\Psi) &\equiv \forall v. \bigvee_i \alpha_{\mathsf{Cart}} \circ \mathsf{hpost}^\#(C_i(v)) && \text{Def. of } \mathsf{post}^\#_{\mathsf{Cart}} \\ &\equiv \forall v. \bigvee_i \bigwedge \{\, p(v) \in \mathcal{P} \mid \mathsf{hpost}^\#(C_i(v)) \models p(v) \,\} && \text{Def. of } \alpha_{\mathsf{Cart}} \\ &\equiv \forall v. \bigvee_i \bigwedge \{\, p(v) \in \mathcal{P} \mid C_i(v) \models \mathsf{hwp}^\#(p(v)) \,\} && \text{by } (*) \ . \end{aligned}$$

Summarizing the above result, the image of a boolean heap Ψ under $\mathsf{post}^\#_{\mathsf{Cart}}$ is constructed by updating for each monomial C_i in Ψ the values of all heap predicates in C_i. These updates are computed by checking for each heap predicate p whether C_i implies the weakest precondition of p or its negation. Hence,

the Cartesian post operator $\mathsf{post}^{\#}_{\mathsf{Cart}}$ corresponds to a boolean heap program as it is defined in Fig. 2. The crucial part in the construction of the boolean heap program lies in the computation of the operator $\mathsf{hwp}^{\#}$. It is implemented using a syntactic operation on formulas (the operator hwp) and by calls to a theorem prover for the entailment tests.

Discussion. Already the first abstraction of the best abstract post operator that we gave above can be formally characterized in terms of a Cartesian abstraction. This leads to a slightly more precise abstraction of $\mathsf{post}^{\#}$; see [14] for details. However, this abstraction is more expensive and introduces a dependency of the operator $\mathsf{hwp}^{\#}$ on the abstract state Ψ for which we compute the post. This dependency violates our goal to have a decoupling of the abstraction phase from the fixed point computation of the analysis.

Focus and Coerce. Cartesian abstraction does not introduce an additional loss of precision as long as the abstract system behaves deterministically, i.e. every abstract object is mapped again to a single abstract object under the operator $\mathsf{hpost}^{\#}$. However, for some commands, e.g. when one iterates over a recursive data structure, the abstract system will behave inherently nondeterministically. In some cases the loss of precision that is caused by this nondeterminism cannot be tolerated. A similar problem occurs in the context of three-valued shape analysis. In [13] so called *focus* and *coerce* operations are used to solve this problem. These operations split three-valued logical structures according to weakest preconditions of predicates and thereby handle the nondeterminism in the abstraction.

Though the *focus* and *coerce* operations are conceptually difficult, it is possible to define a simple corresponding splitting operation in our framework. This operation can be explained in terms of a temporary refinement of the abstract domain. Namely, for splitting one first refines the abstraction by adding new abstraction predicates given by the weakest preconditions of the abstraction predicates that cause the nondeterminism. The refinement causes a splitting of abstract objects and boolean heaps, such that each abstract object in each boolean heap has precise information regarding the weakest preconditions of the problematic predicates. This guarantees that the Cartesian post computes precise updates for these predicates. After computing the Cartesian post the result is mapped back to the original abstract domain by removing the previously added predicates. For a more detailed discussion again see [14].

8 Conclusion

We showed how the abstraction originally proposed in three-valued shape analysis can be constructed in the framework of predicate abstraction. The consequences of our results are:

- a different view on the underlying concepts of three-valued shape analysis.
- a framework of *symbolic* shape analysis. Symbolic means that the abstract post operator is an operation over formulas and is itself constructed solely by deductive reasoning.

– a clear phase separation between the computation of the abstraction and the computation of the fixed point. Among other potential advantages this allows the offline computation of the abstract post operator.
– the possibility to use efficient symbolic methods such as BDDs or SAT solvers. In particular, the abstract post operator itself can be represented as a BDD.

Our framework does not *a priori* impose any restrictions on the data structures implemented by the analyzed programs. Such restrictions only depend on the capabilities of the underlying theorem prover which is used for the entailment tests. There is ongoing research on how to adapt or extend existing theorem provers and decision procedures to the theories that are needed in the context of shape analysis; see e.g. [7,8]. This is a challenging branch for further research.

Another direction for future work is to study refinements of our abstract domain that are even closer to the abstract domain used in three-valued shape analysis. Evidently our framework extends from unary heap predicates to heap predicates of arbitrary arity. If we allow binary relations over heap objects in the abstract domain, we obtain the universal fragment of shape analysis constraints [10].

Extending the abstract domain to the full boolean algebra of shape analysis constraints and thus having exactly the same precision as using three-valued logical structures is more involved. In that case we allow both universally and existentially quantified boolean combinations of heap predicates as base formulas for our abstract domain. Normal forms of conjunctions of these base formulas are an extension of boolean heaps. They correspond to possible boolean coverings of objects with *nonempty* monomials over heap predicates. These normal forms can again be represented as sets of bitvectors over heap predicates. However, in this case it is not clear how Cartesian abstraction can be applied in a way that is similar to the case where we restrict to the universal fragment.

Acknowledgments. We thank Viktor Kuncak, Patrick Lam, and Alexander Malkis for comments and suggestions.

References

1. I. Balaban, A. Pnueli, and L. Zuck. Shape Analysis by Predicate Abstraction. In *Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, LNCS 3385, pages 164–180. Springer, 2005.
2. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Programming language design and implementation (PLDI'01)*, volume 36 of *ACM SIGPLAN Notices*, pages 203–213, 2001.
3. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, LNCS 2031, pages 268–283. Springer, 2001.
4. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages (POPL'79)*, pages 269–282, 1979.

5. D. Dams and K. Namjoshi. Shape Analysis through Predicate Abstraction and Model Checking. In *Verification, Model Checking and Abstract Interpretation (VMCAI'03)*, LNCS 2575, pages 310–323. Springer, 2003.
6. S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification (CAV'97)*, LNCS 1254, pages 72–83. Springer, 1997.
7. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The Boundary Between Decidability and Undecidability for Transitive-Closure Logics. In *Computer Science Logic (CSL 2004)*, LNCS 3210, pages 160–174. Springer, 2004.
8. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Verification Via Structure Simulation. In *Computer Aided Verification (CAV'04)*, LNCS 3114, pages 281–294. Springer, 2004.
9. N. Klarlund and M. Schwartzbach. Graph types. In *Symposium on Principles of Programming Languages (POPL'93)*, pages 196–205, 1993.
10. V. Kuncak and M. Rinard. Boolean Algebra of Shape Analysis Constraints. In *Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, LNCS 2937, pages 59–72. Springer, 2004.
11. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, LNCS 3385, pages 181–198. Springer, 2005.
12. A. Møller and M. Schwartzbach. The pointer assertion logic engine. In *Programming language design and implementation (PLDI'01)*, pages 221–231, 2001.
13. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.
14. T. Wies. Symbolic Shape Analysis. Diploma thesis, Universität des Saarlandes, Saarbrücken, Germany, 2004.
15. G. Yorsh. Logical Characterizations of Heap Abstractions. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2003.
16. G. Yorsh, T. Reps, and M. Sagiv. Symbolically Computing Most-Precise Abstract Operations for Shape Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, LNCS 2988, pages 530–545. Springer, 2004.

Interprocedural Shape Analysis for Cutpoint-Free Programs

Noam Rinetzky[1,⋆], Mooly Sagiv[1], and Eran Yahav[2]

[1] Tel Aviv University
{maon, msagiv}@tau.ac.il
[2] IBM T.J. Watson Research Center
eyahav@us.ibm.com

Abstract. We present a framework for interprocedural shape analysis, which is context- and flow-sensitive with the ability to perform destructive pointer updates. We limit our attention to cutpoint-free programs—programs in which reasoning on a procedure call only requires consideration of context reachable from the actual parameters. For such programs, we show that our framework is able to perform an efficient modular analysis. Technically, our analysis computes procedure summaries as transformers from inputs to outputs while *ignoring parts of the heap not relevant to the procedure*. This makes the analysis modular in the heap and thus allows reusing the effect of a procedure at different call-sites and even between different contexts occurring at the same call-site. We have implemented a prototype of our framework and used it to verify interesting properties of cutpoint-free programs, including partial correctness of a recursive quicksort implementation.

1 Introduction

Shape-analysis algorithms statically analyze a program to determine information about the heap-allocated data structures that the program manipulates. The algorithms are *conservative* (sound), i.e., the discovered information is true for every input. Handling the heap in a precise manner requires strong pointer updates [6]. However, performing strong pointer updates requires flow-sensitive context-sensitive analysis and expensive heap abstractions that may be doubly-exponential in the program size [36]. The presence of procedures escalates the problem because of interactions between the program stack and the heap [34] and because recursive calls may introduce exponential factors in the analysis. This makes interprocedural shape analysis a challenging problem.

This paper introduces a new approach for shape analysis for a class of imperative programs. The main idea is to restrict the "sharing patterns" occurring in procedure calls. This allows procedures to be analyzed ignoring the part of the heap not reachable from actual parameters. Moreover, shape analysis can conservatively detect violations of the above restrictions, thus allowing to treat existing programs. A prototype of this approach was implemented and used to verify properties that could not be automatically verified before, including the partial correctness of a recursive quicksort [16] implementation (i.e., show that it returns an ordered permutation of its input).

⋆ This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant No 304/03).

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 284–302, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Our restriction on programs is inspired by [33]. There, Rinetzky et. al. present a non-standard semantics for arbitrary programs in which procedures operate on local heaps containing only the objects reachable from actual parameters. The most complex aspect of [33] is the treatment of sharing between the local heap and the rest of the heap. The problem is that the local heap can be accessed via access paths which bypass actual parameters. Therefore, objects in the local heap are treated differently when they separate the local heap (that can be accessed by a procedure) from the rest of the heap (which—from the viewpoint of that procedure—is non-accessible and immutable). We call these objects *cutpoints* [33]. We refer to an invocation in which no such cutpoint object exists as a *cutpoint-free invocation*. We refer to an execution of a program in which all invocations are cutpoint-free as a *cutpoint-free execution*, and to a program in which all executions are cutpoint-free as a *cutpoint-free program*. (We define these notions more formally in the following sections).

While many programs are not cutpoint-free, we observe that a reasonable number of programs, including all examples used in [13, 34, 19] are cutpoint-free, as well as many of the programs in [12, 37]. One of the key observations in this paper, is that we can exploit cutpoint-freedom to construct an interprocedural shape analysis algorithm that efficiently reuses procedure summaries.

In this paper, we present $\mathcal{LCPF}$, an operational semantics that efficiently handles cutpoint-free programs. This semantics is interesting because procedures operate on local heaps, thus supporting the notion of heap-modularity while permitting the usage of a global heap and destructive updates. Moreover, the absence of cutpoints drastically simplifies the meaning of procedure calls. $\mathcal{LCPF}$ checks that a program execution is indeed cutpoint-free and halts otherwise. As a result, it is applicable to any arbitrary program, and does not require an a priori classification of a program as cutpoint-free. We show that for cutpoint-free programs, $\mathcal{LCPF}$ is observationally equivalent to the standard global-heap semantics.

$\mathcal{LCPF}$ gives rise to an efficient interprocedural shape-analysis for cutpoint-free programs. Our interprocedural shape-analysis is a functional interprocedural analysis [10, 38, 20, 29, 11, 19, 2]. It tabulates abstractions of memory states before and after procedure calls. However, memory states are represented in a non-standard way *ignoring parts of the heap not relevant to the procedure*. This reduces the complexity of the analysis because the analysis of procedures does not represent information on references and on the heap from calling contexts. Indeed, this makes the analysis modular in the heap and thus allows reusing the summarized effect of a procedure at different calling contexts. Finally, this reduces the asymptotic complexity of the interprocedural shape analysis. For programs without global variables, the worst case time complexity of the analysis is doubly-exponential in the maximum number of local variables in a procedure, instead of being doubly-exponential in the total number of local variables [34].

Technically, our algorithm is built on top of the 3-valued logical framework for program analysis of [23, 36]. Thus, it is parametric in the heap abstraction and in the concrete effects of program statements, allowing to experiment with different instances of interprocedural shape analyzers. For example, we can employ different abstractions for

singly-, doubly-linked lists, and trees. Also, a combination of theorems in [35] and [36] guarantees that every instance of our *interprocedural* framework is sound (see Sec. 3).

This paper also provides an initial empirical evaluation of our algorithm. Our empirical evaluation indicates that the analysis is precise enough to prove properties such as the absence of null dereferences, preservation of data structure invariants such as list-ness, tree-ness, and sorted-ness for iterative and recursive programs with deep references into the heap and destructive updates. We observe that the cost of analyzing recursive procedures is comparable to the cost of analyzing their iterative counterparts. Moreover, the cost of analyzing a program with procedures is smaller than the cost of analyzing the same program with procedure bodies inlined.

```
public class List{
   List n = null;
   int data;
   public List(int d){
      this.data = d;
   }
   static public List create3(int k) {
      List t1 = new List(k), t2 = new List(k+1), t3 = new List(k+2);
      t1.n = t2; t2.n = t3;
      return t1;
   }
   public static List splice(List p, List q) {
      List w = q;
      if (p != null) {
         List pn = p.n;
         p.n = null;
         p.n = splice(q, pn);
         w = p;
      }
      return w;
   }
   public static void main(String[] argv) {
      List x = create3(1), y = create3(4), z = create3(7);
      List t = splice(x, y);
      List s = splice(y, z);
   }
}
```

Fig. 1. A Java program recursively splicing three singly-linked lists using destructive updates

1.1 Main Results

The contributions of this paper can be summarized as follows:

1. We define the notion of cutpoint-free programs, in which reasoning about a procedure allows ignoring the context not reachable from its actual parameters.
2. We show that interesting cutpoint-free programs can be written naturally, e.g., programs manipulating unshared trees and a recursive implementation of quicksort. We also show that some interesting existing programs are cutpoint-free, e.g., all programs verified using shape analysis in [13,34,19], and many of those in [12,37].
3. We define an operational semantics for arbitrary Java-like programs that verifies that a program execution is cutpoint free. In this semantics, procedures operate on

local heaps, thus supporting the notion of heap-modularity while permitting the usage of a global heap and destructive updates.
4. We present an interprocedural shape analysis for cutpoint-free programs. Our analysis is modular in the heap and thus allows reusing the effect of a procedure at different calling contexts and at different call-sites. Our analysis goes beyond the limits of existing approaches and was used to verify a recursive quicksort implementation.
5. We implemented a prototype of our approach. Preliminary experimental results indicate that: (i) the cost of analyzing recursive procedures is similar to the cost of analyzing their iterative versions; (ii) our analysis benefits from procedural abstraction; (iii) our approach compares favorably with [34, 19].

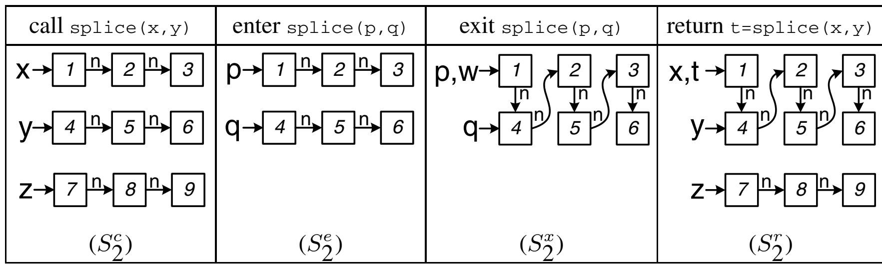


Fig. 2. Concrete states for the invocation `t = splice(x, y)` in the running example

1.2 Motivating Example

Fig. 1 shows a simple Java program that splices three unshared, disjoint, acyclic singly-linked lists using a recursive `splice` procedure. This program serves as a running example in this paper.

For each invocation of `splice`, our analyzer verifies that the returned list is acyclic and not heap-shared;[1] that the first parameter is aliased with the returned reference; and that the second parameter points to the second element in the returned list.

For this example, our algorithm effectively reuses procedure summaries, and only analyzes `splice(p,q)` once for every possible abstract input. As shown in Sec. 3.3, this means that `splice(p,q)` will be only analyzed a total number of 9 times. This should be contrasted with [34], in which no summaries are computed, and the procedure is analyzed 66 times. Compared to [19], our algorithm can summarize procedures in a more compact way (see Sec. 5).

1.3 Local Heaps, Relevant Objects, Cutpoints, and Cutpoint-Freedom

In our semantics, procedures operate on local heaps. The local heap contains only the part of the program's heap accessible to the procedure. Thus, procedures are invoked on local heaps containing only objects reachable from actual parameters. We refer to these objects as the *relevant* objects for the invocation.

[1] An object is heap-shared if it is pointed-to by a field of more than one object.

Example 1. Fig. 2 shows the concrete memory states that occur at the call `t=splice (x,y)`. S_2^c shows the state at the point of the call, and S_2^e shows the state on entry to `splice`. Here, `splice` is invoked on local heap containing the (relevant) objects reachable from either `x` or `y`.

The fact that the local heap of the invocation `t=splice(x,y)` contains only the lists referenced by `x` and `y`, guarantees that destructive updates performed by `splice` can only affect access paths that pass through an object referenced by either `x` or `y`. Similarly, the invocation `s=splice(y,z)` in the concrete memory state S_3^c, shown in Fig. 3(a), can only affect access paths that pass through an object referenced by either `y` or `z`.

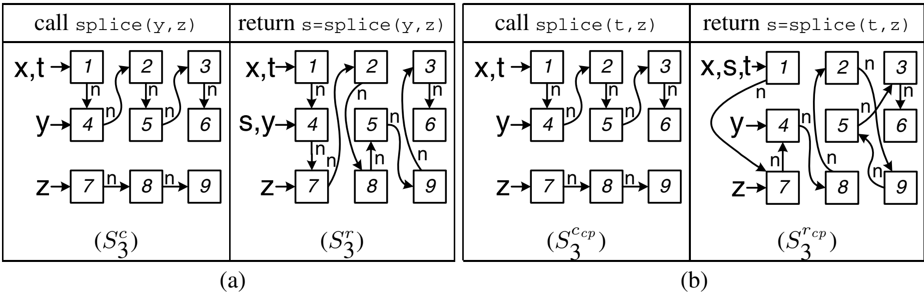


Fig. 3. Concrete states for: (a) the invocation `s=splice(y,z)` in the program of Fig. 1; (b) a variant of this program with an invocation `s=splice(t,z)`

Obviously, this is not always the case. For example, consider a variant of the example program in which the second call `s=splice(y,z)` is replaced by a call `s=splice(t,z)`. $S_3^{c_{cp}}$ and $S_3^{r_{cp}}$, depicted in Fig. 3(b), show the concrete states when `s=splice(t,z)` is invoked and when it returns, respectively. As shown in the figure, the destructive updates of the `splice` procedure change not only paths from `t` and `z`, but also change the access paths from `y`.

A *cutpoint* for an invocation is an object which is: (i) reachable from an actual parameter, (ii) not pointed-to by an actual parameter, and (iii) reachable without going through an object which is pointed-to by an actual parameter (that is, it is either pointed-to by a variable or by an object not reachable from the parameters). In other words, a cutpoint is a relevant object that separates the part of the heap which is passed to the callee from the rest of the heap, but which is not pointed-to by a parameter. The object pointed-to by `y` at the call `s=splice(t,z)` (Fig. 3(b)) is a *cutpoint*, and this invocation is not *cutpoint-free*. In contrast, the call `t=splice(x,y)` (Fig. 2) does not have any cutpoints and is therefore *cutpoint-free*. In fact, all invocations in the program of Fig. 1, including recursive ones, are cutpoint-free, and the program is a cutpoint-free program.

Our analyzer verifies that the running example is a cutpoint-free program. It also detects that in the variant of our running example, the call `s=splice(t,z)` is not a cutpoint-free invocation.

1.4 Outline

The rest of the paper is organized as follows. Sec. 2 defines our local heap concrete semantics. Sec. 3 conservatively abstracts this semantics, providing a heap-modular interprocedural shape analysis algorithm. Sec. 4 describes our implementation and experimental results. Sec. 5 describes related work, and Sec. 6 concludes. Due to space limitations, formal details and more experimental results appear in [35].

2 Concrete Semantics

In this section, we present $\mathcal{LCPF}$, a large-step concrete semantics that serves as the basis for our abstraction. In $\mathcal{LCPF}$, an invoked procedure is passed only relevant objects. $\mathcal{LCPF}$ has two novel aspects: (i) it verifies that the execution is cutpoint-free; (ii) it has a *simple* rule for procedure calls that exploits (the verified) cutpoint-freedom. Nevertheless, in [35], we show that for cutpoint-free programs $\mathcal{LCPF}$ is observationally equivalent to a standard store-based global-heap semantics. For simplicity, $\mathcal{LCPF}$ only keeps track of pointer-valued variables and fields.

Table 1. Predicates used in the concrete semantics

Predicate	Intended Meaning
$T(v)$	v is an object of type `T`
$f(v_1, v_2)$	the `f`-field of object v_1 points to object v_2
$eq(v_1, v_2)$	v_1 and v_2 are the same object
$x(v)$	reference variable `x` points to the object v
$inUc(v)$	v originates from the caller's memory state at the call site
$inUx(v)$	v originated from the callee's memory state at the exit site

2.1 Concrete Memory States

We represent memory states using 2-valued logical structures. A 2-valued logical structure over a set of predicates $\mathcal{P}$ is a pair $S = \langle U^S, \iota^S \rangle$ where:

- U^S is the universe of the 2-valued structure. Each individual in U^S represents a heap-allocated object.
- ι^S is an interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in \mathcal{P}$ of arity k, $\iota^S(p) : {U^S}^k \to \{0, 1\}$. Predicates correspond to tracked properties of heap-allocated objects.

The set of *2-valued* logical structures is denoted by $\mathit{2Struct}$.

In the rest of the paper, we assume to be working with a fixed arbitrary program P. The program P consists of a collection of types, denoted by $\mathit{TypeId}^\star$. The set of all reference fields defined in P is denoted by $\mathit{FieldId}^\star$. For a procedure p, V_p denotes the set of its local reference variables, including its formal parameters. The set of all the local (reference) variables in P is denoted by $\mathit{Local}^\star$. For simplicity, we assume

formal parameters are not assigned and that p always returns a value using a designated variable $ret_p \in V_p$. For example, $ret_{\texttt{splice}} = \texttt{w}$.

Tab. 1 shows the core predicates used in this paper. A unary predicate $T(v)$ holds for heap-allocated objects of type $\texttt{T} \in \mathit{TypeId}^\star$. A binary predicate $f(v_1, v_2)$ holds when the $\texttt{f} \in \mathit{FieldId}^\star$ field of v_1 points-to v_2. The designated binary predicate $eq(v_1, v_2)$ is the equality predicate recording equality between v_1 and v_2. A unary predicate $x(v)$ holds for an object that is pointed-to by the reference variable $\texttt{x} \in \mathit{Local}^\star$ of the *current* procedure.[2] The role of the predicates $inUc$ and $inUx$ is explained in Sec. 2.2.

2-valued logical structures are depicted as directed graphs. We draw individuals as boxes. We depict the value of a pointer variable x by drawing an edge from x to the individual that represent the object that x points-to. For all other unary predicates p, we draw p inside a node u when $\iota^S(p)(u) = 1$; conversely, when $\iota^S(p)(u) = 0$ we do not draw p in u. A directed edge between nodes u_1 and u_2 that is labeled with a binary predicate symbol p indicates that $\iota^S(p)(u_1, u_2) = 1$. For clarity, we do not draw the unary *List* predicate, and the binary equality predicate eq.

Example 2. The structure S_2^c of Fig. 2 shows a *2-valued* logical structure that represents the memory state of the program at the call `t=splice(x, y)`. The depicted numerical values are only shown for presentation reasons, and have no meaning in the logical representation.

2.2 Inference Rules

The meaning of statements is described by a transition relation $\overset{lcpf}{\rightsquigarrow} \subseteq (\mathit{2Struct} \times st) \times \mathit{2Struct}$ that specifies how a statement st transforms an incoming logical structure into an outgoing logical structure. For assignments, this is done primarily by defining the values of the predicates in the outgoing structure using first-order logic formulae with transitive closure over the incoming structure [36]. The inference rules for assignments are rather straightforward and can be found in [35]. For control statements, we use the standard rules of natural semantics, e.g., see [26].

Our treatment of procedure call and return could be briefly described as follows: (i) the call rule is applied, first checking that the invocation is cutpoint-free (by evaluating the side condition), and (ii) proceeding to construct the memory state at the callee's entry site (S_e) if the side condition holds; (iii) the caller's memory state at the call site (S_c) and the callee's memory state at the exit site (S_x) are used to construct the caller's memory state at the return site (S_r). We now formally define and explain these steps.

Fig. 4 specifies the procedure call rule for an arbitrary call statement $y = p(x_1, \ldots, x_k)$ by an arbitrary function q. The rule is instantiated for each call statement in the program.

Verifying Cutpoint-Freedom. The semantics uses the side condition of the procedure call rule to ensure that the execution is cutpoint-free. The side condition asserts that no object is a cutpoint. This is achieved by verifying that the formula $isCP_{q,\{x_1,\ldots,x_k\}}(v)$,

[2] For simplicity, we use the same set of predicates for all procedures. Thus, our semantics ensures that $\iota^S(x) = \lambda u.0$ for every local variable x that does not belong to the current call.

defined in Tab. 2, does not hold for any object at S_c, the memory state that arises when $p(x_1, \ldots, x_k)$ is invoked by q.

The formula $isCP_{q,\{x_1,\ldots,x_k\}}(v)$, holding when v is a cutpoint object, is comprised of three conjuncts. The first conjunct, requires that v be reachable from an actual parameter. The second conjunct, requires that v not be pointed-to by an actual parameter. The third conjunct, requires that v be an entry point into p's local heap, i.e., is pointed-to by a local variable of q (the caller procedure) or by a field of an object not passed to p.

Example 3. The structure S^c_2 of Fig. 2 depicts the memory state at the point of the call `t = splice(x, y)`. In this state, the formula $isCP_{main,\{x,y\}}(v)$ does not hold for any object. On the other hand, when `s = splice(t, z)` is invoked at $S^{C_{cp}}_3$ of Fig. 3(b), the object pointed-to by `y` is a cutpoint. Note, that the formula $isCP_{main,\{t,z\}}(v)$ evaluates to 1 when v is bound to this object: the formula $R_{\{t,z\}}(v)$ holds for every object in `t`'s list. In particular, it holds for the second object which is pointed-to by a local variable (`y`) but not by an actual parameter (`t`, `z`).

Note that $\mathcal{LCPF}$ considers only the values of variables that belong to the current call when it detects cutpoints. This is possible because all pending calls are cutpoint-free. This greatly simplifies the cutpoint detection compared to [33].

Computing the Memory State at the Entry Site. S_e, the memory state at the entry site to p, represents the local heap passed to p. It contains only these individuals in S_c that represent objects that are relevant for the invocation. The formal parameters are initialized by $updCall_q^{y=p(x_1,\ldots,x_k)}$, defined in Fig. 5(a). The latter, specifies the value of the predicates in S_e using a predicate-update formulae evaluated over S_c. We use the convention that the updated value of x is denoted by x'. Predicates whose update formula is not specified, are assumed to be unchanged, i.e., $x'(v_1, \ldots) = x(v_1, \ldots)$. Note that only the predicates that represent variable values are modified. In particular, field values, represented by binary predicates, remain in p's local heap as in S_c.

Table 2. Formulae shorthands and their intended meaning

Shorthand	**Formula**	**Intended Meaning**
$F(v_1, v_2)$	$\bigvee_{f \in FieldId^{\star}_P} f(v_1, v_2)$	v_1 has a field that points to v_2
$\varphi^*(v_1, v_2)$	$eq(v_1, v_2) \lor$ $(TC\ w_1, w_2 : \varphi(w_1, w_2))(v_1, v_2)$	the reflexive transitive closure of φ
$R_{\{x_1,\ldots,x_k\}}(v)$	$\bigvee_{x \in \{x_1,\ldots,x_k\}} \exists v_1 : x(v_1) \land F^*(v_1, v)$	v is reachable from $\mathtt{x}_1$ or ... or $\mathtt{x}_k$
$isCP_{q,\{x_1,\ldots,x_k\}}(v)$	$R_{\{x_1,\ldots,x_k\}}(v) \land$ $(\neg x_1(v) \land \ldots \land \neg x_k(v)) \land$ $(\bigvee_{y \in V_q} y(v) \lor$ $\exists v_1 : \neg R_{\{x_1,\ldots,x_k\}}(v_1) \land F(v_1, v))$	v is a cutpoint

Example 4. The structure S^e_2 of Fig. 2 depicts the memory state at the entry site to `splice` when `t = splice(x, y)` is invoked at the memory state S^c_2. Note that the list referenced by `z` is not passed to `splice`. Also note that the element which was referenced by `x` is now referenced by `p`. This is the result of applying the update formula $p'(v) = x(v)$ for the predicate p in this call. Similarly, the element which was referenced by `y` is now referenced by `q`.

$$\frac{\langle \textit{body of } p, S_e\rangle \overset{lcpf}{\rightsquigarrow} S_x}{\langle y = p(x_1,\ldots,x_k), S_c\rangle \overset{lcpf}{\rightsquigarrow} S_r} \; S_c \models \forall v\colon \neg isCP_{q,\{x_1,\ldots,x_k\}}(v)$$

where

$S_e = \langle U_e, \iota_e\rangle$ where
$U_e = \{u \in U^{S_c} \mid S_c \models R_{\{x_1,\ldots,x_k\}}(u)\}$
$\iota_e = updCall_q^{y=p(x_1,\ldots,x_k)}(S_c)$

$S_r = \langle U_r, \iota_r\rangle$ where
Let $U' = \{u.c \mid u \in U_c\} \cup \{u.x \mid u \in U_x\}$

$$\iota' = \lambda p \in \mathcal{P}. \begin{cases} \iota_c[inUc \mapsto \lambda v.1](p)(u_1,\ldots,u_m) & : u_1 = w_1.c, \ldots, u_m = w_m.c \\ \iota_x[inUx \mapsto \lambda v.1](p)(u_1,\ldots,u_m) & : u_1 = w_1.x, \ldots, u_m = w_m.x \\ 0 & : otherwise \end{cases}$$

in $U_r = \{u \in U' \mid \langle U', \iota'\rangle \not\models inUc(u) \wedge R_{\{x_1,\ldots,x_k\}}(u)\}$
$\iota_r = updRet_q^{y=p(x_1,\ldots,x_k)}(\langle U', \iota'\rangle)$

Fig. 4. The inference rule for a procedure call $y = p(x_1,\ldots,x_k)$ by a procedure q. The functions $updCall_q^{y=p(x_1,\ldots,x_k)}$ and $updRet_q^{y=p(x_1,\ldots,x_k)}$ are defined in Fig. 5.

a. Predicate update formulae for $updCall_q^{y=p(x_1,\ldots,x_k)}$

$$z'(v) = \begin{cases} x_i(v) & : z = h_i \\ 0 & : z \in Local^\star \setminus \{h_1,\ldots,h_k\} \end{cases}$$

b. Predicate update formulae for $updRet_q^{y=p(x_1,\ldots,x_k)}$

$$z'(v) = \begin{cases} ret_p(v) & : z = y \\ inUc(v) \wedge z(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee & : z \in V_q \setminus \{y\} \\ \quad \exists v_1\colon z(v_1) \wedge match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1,v) & \\ 0 & : z \in Local^\star \setminus V_q \end{cases}$$

$$\begin{aligned} f'(v_1,v_2) = {} & inUx(v_1) \wedge inUx(v_2) \wedge f(v_1,v_2) \vee \\ & inUc(v_1) \wedge inUc(v_2) \wedge f(v_1,v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \vee \\ & inUc(v_1) \wedge inUx(v_2) \wedge \exists v_{sep}\colon f(v_1,v_{sep}) \wedge match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_{sep},v_2) \end{aligned}$$

$inUc'(v) = inUx'(v) = 0$

Fig. 5. Predicate-update formulae for the core predicates used in the procedure call rule. We assume that the p's formal parameters are $h_1,\ldots,h_k$. There is a separate update formula for every local variable $\mathtt{z} \in Local^\star$ and for every field $\mathtt{f} \in FieldId^\star$.

Computing the Memory State at the Return Site. The memory state at the return-site (S_r) is constructed as a combination of the memory state in which p was invoked (S_c) and the memory state at p's exit-site (S_x). Informally, S_c provides the information about the (unmodified) irrelevant objects and S_x contributes the information about the destructive updates and allocations made during the invocation.

The main challenge in computing the effect of a procedure is relating the objects at the call-site to the corresponding objects at the return site. The fact that the invocation

is cutpoint-free guarantees that the only references into the local heap are references to objects referenced by an actual parameter. This allows us to reflect the effect of p into the local heap of q by: (i) replacing the relevant objects in S_c with S_x, the local heap at the exit from p; (ii) redirecting all references to an object referenced by an actual parameter to the object referenced by the corresponding formal parameter in S_x.

Technically, S_c and S_x are *combined* into an intermediate structure $\langle U', \iota' \rangle$. The latter contains a copy of the memory states at the call site and at the exit site. To distinguish between the copies, the auxiliary predicates $inUc$ and $inUx$ are set to hold for individuals that originate from S_c and S_x, respectively. Pointer redirection is specified by means of predicate update formulae, as defined in Fig. 5(b). The most interesting aspect of these update-formulae is the formula $match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}$, defined below:

$$match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1,v_2) \stackrel{\text{def}}{=} \bigvee_{i=1}^{k} inUc(v_1) \wedge x_i(v_1) \wedge inUx(v_2) \wedge h_i(v_2)$$

This formula matches an individual that represents an object which is referenced by an actual parameter at the call-site, with the individual that represents the object which is referenced by the corresponding formal parameter at the exit-site. Our assumption that formal parameters are not modified allows us to match these two individuals as representing the same object. Once pointer redirection is complete, all individuals originating from S_c and representing relevant objects are removed, resulting with the updated memory state of the caller.

Example 5. S_2^c and S_2^x, shown in Fig. 2, represent the memory states at the call-site and at the exit-site of the invocation `t=splice(x,y)`, respectively. Their combination according to the procedure call rule is S_2^r, which represents the memory state at the return site. Note that the lists of `x` and `y` from the call-site were replaced by the lists referenced by `p` and `q`. The list referenced by `z` was taken as is from the call-site.

Table 3. The instrumentation predicates used in this paper

Predicate	Intended Meaning	Defining Formula
$r_{obj}(v_1,v_2)$	v_2 is reachable from v_1 by some field path	$F^*(v_1,v_2)$
$ils(v)$	v is *locally* shared. i.e., v is pointed-to by a field of more than one object in the *local-heap*	$\exists v_1,v_2 : \neg eq(v_1,v_2) \wedge F(v_1,v) \wedge F(v_2,v)$
$c(v)$	v resides on a directed cycle of fields	$\exists v_1 : F(v,v_1) \wedge F^*(v_1,v)$
$r_x(v)$	v is reachable from variable `x`	$\exists v_x : x(v_x) \wedge F^*(v_x,v)$

3 Abstract Semantics

In this section, we present $\mathcal{LCPF}^{\#}$, a conservative abstract semantics abstracting $\mathcal{LCPF}$.

3.1 Abstract Memory States

We conservatively represent multiple concrete memory states using a single logical structure with an extra truth-value $1/2$ which denotes values which may be 1 and which may be 0. The information partial order on the set $\{0, 1/2, 1\}$ is defined as $0 \sqsubseteq 1/2 \sqsupseteq 1$, and $0 \sqcup 1 = 1/2$.

An *abstract state* is a 3-valued logical structure $S^\sharp = \langle U^{S^\sharp}, \iota^{S^\sharp} \rangle$ where:

- $U^{S^\sharp}$ is the universe of the structure. Each individual in $U^{S^\sharp}$ possibly represents many heap-allocated objects.
- $\iota^{S^\sharp}$ is an interpretation function mapping predicates to their truth-value in the structure, i.e., for every predicate $p \in \mathcal{P}$ of arity k, $\iota^S(p) \colon {U^{S^\sharp}}^k \to \{0, 1/2, 1\}$.

The set of *3-valued* logical structures is denoted by $\mathit{3Struct}$.

Instrumentation Predicates. Instrumentation predicates record derived properties of individuals, and are defined using a logical formula over core predicates. Instrumentation predicates are stored in the logical structures like core predicates. They are used to refine the abstract semantics, as we shall shortly see. Tab. 3 lists the instrumentation predicates used in this paper.

Canonical Abstraction. We now formally define how concrete memory states are represented using abstract memory states. The idea is that each individual from the (concrete) state is mapped into an individual in the abstract state. An abstract memory state may include *summary nodes*, i.e., an individual which corresponds to one or more individuals in a concrete state represented by that abstract state.

A *3-valued* logical structure $S^\sharp$ is a **canonical abstraction** of a *2-valued* logical structure S if there exists a surjective function $f \colon U^S \to U^{S^\sharp}$ satisfying the following conditions: (i) For all $u_1, u_2 \in U^S$, $f(u_1) = f(u_2)$ iff for all unary predicates $p \in \mathcal{P}$, $\iota^S(p)(u_1) = \iota^S(p)(u_2)$, and (ii) For all predicates $p \in \mathcal{P}$ of arity k and for all k-tuples $u_1^\sharp, u_2^\sharp, \ldots, u_k^\sharp \in U^{S^\sharp}$,

$$\iota^{S^\sharp}(p)(u_1^\sharp, u_2^\sharp, \ldots, u_k^\sharp) = \bigsqcup_{\substack{u_1, \ldots, u_k \in U^S \\ f(u_i) = u_i^\sharp}} \iota^S(p)(u_1, u_2, \ldots, u_k).$$

The set of concrete memory states such that $S^\sharp$ is their canonical abstraction is denoted by $\gamma(S^\sharp)$. Finally, we say that a node $u^\sharp \in U^{S^\sharp}$ **represents** node $u \in U$, when $f(u) = u^\sharp$. Note that *only* for a summary node u, $\iota^{S^\sharp}(eq)(u, u) = 1/2$.

3-valued logical structures are also drawn as directed graphs. Definite values (0 and 1) are drawn as for 2-valued structures. Binary indefinite predicate values ($1/2$) are drawn as dotted directed edges. Summary nodes are depicted by a double frame.

Example 6. Fig. 6 shows the abstract states (as *3-valued* logical structures) representing the concrete states of Fig. 2. Note that only the local variables `p` and `q` are represented inside the call to `splice(p,q)`. Representing only the local variables inside a call ensures that the number of unary predicates to be considered when analyzing the procedure is proportional to the number of its local variables. This reduces the overall com-

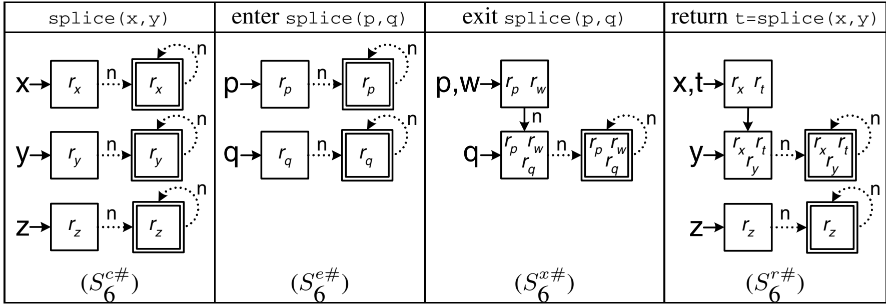


Fig. 6. Abstract states for the invocation `t = splice(x, y);` in the running example

plexity of our algorithm to be worst-case doubly-exponential in the maximal number of local variables rather than doubly-exponential in their total number (as in e.g., [34]).

The Importance of Reachability. Recording derived properties by means of *instrumentation predicates* may provide additional information that would have been otherwise lost under abstraction. In particular, because canonical abstraction is directed by unary predicates, adding unary instrumentation predicates may further refine the abstraction. This is called the *instrumentation principle* in [36]. In our framework, the predicates that record reachability from variables plays a central role. They enable us to identify the individuals representing objects that are reachable from actual parameters. For example, in the *3-valued* logical structure $S_6^{c\#}$ depicted in Fig. 6, we can detect that the top two lists represent objects that are reachable from the actual parameters because either r_x or r_y holds for these individuals. None of these predicates holds for the individuals at the (irrelevant) list referenced by `z`. We believe that these predicates should be incorporated in any instance of our framework.

3.2 Inference Rules

The meaning of statements is described by a transition relation $\overset{lcpf^\#}{\rightsquigarrow} \subseteq (\mathit{3Struct} \times st) \times \mathit{3Struct}$. Because our framework is based on [36], the specification of the concrete operational semantics for program statements (as transformers of 2-valued structures) in Sec. 2, also defines the corresponding abstract semantics (as transformers of 3-valued structures). This abstract semantics is obtained by reinterpreting logical formulae using a 3-valued logic semantics and serves as the basis for an abstract interpretation. In particular, reinterpreting the side condition of the procedure call rule conservatively, verifies that the *program* is cutpoint free. In this paper, we directly utilize the implementation of these ideas available in TVLA [23].

In principle, the effect of a statement on the values of the instrumentation predicates can be evaluated using their defining formulae and the update formulae for the core predicates. In practice, this may lead to imprecise results in the analysis. It is far better to supply the update formula for the instrumentation predicates too. In this paper, we manually provide the update formulae of the instrumentation predicates (as done e.g., in [36, 22, 34]). Automatic derivation of update formulae for the instrumentation

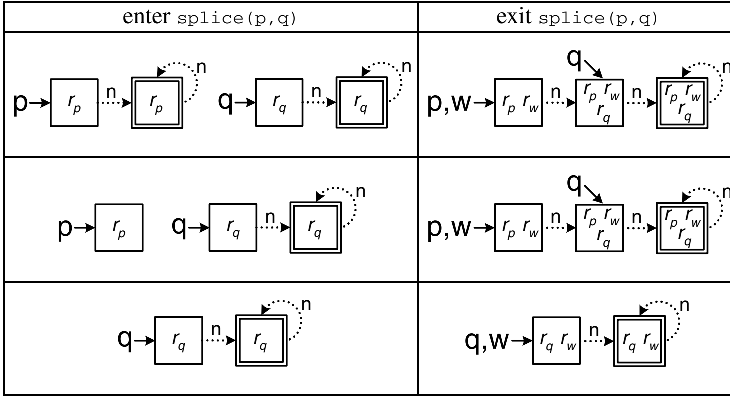


Fig. 7. Partial tabulation of abstract states for the splice procedure

predicates [30] is currently not implemented in our framework. We note that update formulae are provided at the level of the programming language, and are thus applicable to arbitrary procedures and programs. Predicate update-formulae for the instrumentation predicates are provided in [35].

The soundness of our abstract semantics is guaranteed by the combination of the theorems in [35] and [36]:

- In [35], we show that for cutpoint-free programs $\mathcal{LCPF}$ is observationally equivalent to a standard store-based global-heap semantics.
- In [36], it is shown that every program-analyzer which is an instance of their framework is sound with respect to the concrete semantics it is based on.

3.3 Interprocedural Functional Analysis via Tabulation of Abstract Local Heaps

Our algorithm computes procedure summaries by tabulating input abstract memory-states to output abstract memory-states. The tabulation is restricted to abstract memory-states that occur in the *analyzed* program. The tabulated abstract memory-states represent local heaps, and are therefore independent of the context in which a procedure is invoked. As a result, the summary computed for a procedure could be used at different calling contexts and at different call-sites.

Our interprocedural analysis algorithm is a variant of the IFDS-framework [29] adapted to work with local-heaps. The main difference between our framework and [29] is in the way return statements are handled: In [29], the dataflow facts that reach a return-site come either from the call-site (for information pertaining to local variables) or from the exit-site (for information pertaining to global variables). In our case, the information about the heap is obtained by *combining* pair-wise the abstract memory states at the call-site with their counterparts at the exit-site. A detailed description of our tabulation algorithm can be found in [35].

Example 7. Fig. 7 shows a partial tabulation of abstract local heaps for the `splice` procedure of the running example. The figure shows 3 possible input states of the list

pointed-to by `p`. Identical possible input states of the list pointed-to by `q`, and their combinations are not shown. As mentioned in Sec. 1, the splice procedure is only analyzed 9 times before its tabulation is complete, producing a summary that is then reused whenever the effect of `splice(p, q)` is needed.

4 Prototype Implementation

We have implemented a prototype of our framework using TVLA [23]. The framework is parametric in the heap-abstraction and in the operational semantics. We have instantiated the framework to produce a shape-analysis algorithm for analyzing Java programs that manipulate (sorted) singly-linked lists and unshared trees.

The join operator in our framework can be either set-union or a more "aggressive" partial-join operation [24]. The former ensures that the analysis is fully-context sensitive. The latter exploits the fact that our abstract domain has a Hoare order and returns an upper approximation of the set-union operator. Our experiments were conducted with the partial-join operator.

Our analysis was able to verify that all the tested programs are cutpoint-free and *clean*, i.e., do not perform null-dereference and do not leak memory. For singly-linked-list-manipulating programs (Tab. 4.a), we also verified that the invoked procedures preserve list acyclicity. The analysis of the tree-manipulating programs (Tab. 4.b) verified that the tree invariants hold after the procedure terminates. For these programs we assume (and verify) that the trees are unshared. The analysis of the sorting programs (Tab. 4.c) verified that the sorting procedure returns a sorted permutation of its input list. To prove this property we adapted the abstraction used in [22]. We note that prior attempts to verify the partial correctness of `quicksort` using TVLA were not successful. For more details, see [35].

For two of our example programs (`quicksort` and `reverse8`), cutpoints were created as a result of objects pointed-to by a dead variable or a dead field at the point of a call. We manually rewrote these programs to eliminate these (false) cutpoints.

Tab. 4a-c compares the cost of analysis for iterative and recursive implementations of a given program.[3] For these programs, we found that the cost of analyzing recursive procedures and iterative procedures is comparable in most cases. We note that our tests were of *client* programs and not a single procedure, i.e., in all tests, the program also allocates the data structure that it manipulates.

Tab. 4.d shows that our approach compares favorably with existing TVLA-based interprocedural shape analyzers [34,19]. The experiments measure the cost of analyzing 4 recursive procedures that manipulate singly linked lists. For fair comparison with [33] and [18], we follow them and do not measure the cost of list allocation in these tests. All analyzers successfully verified that these (correct) procedures are clean and preserve list acyclicity. [19] was able to prove that `reverse` reverses the list and to pinpoint the location in the list that `delete` removed an element from. However, the cost of analysis for `insert` and `delete` in [19] was higher than the cost in [34] and in our analysis.

[3] `revApp` is a recursive procedure. We analyzed it once with an iterative append procedure and once with a recursive append. Tail sort is a recursive procedure. We analyzed it once with an iterative insert procedure and once with a recursive insert.

Table 4. Experimental results. Time is measured in seconds. Space is measured in megabytes. Experiments performed on a machine with a 1.5 Ghz Pentium M processor and 1 Gb memory.

Iterative vs. Recursive Programs				
Implementation	**Iterative**		**Recursive**	
a. List manipulating programs	**Space**	**Time**	**Space**	**Time**
create creates a list	2.5	11.5	2.3	9.3
find searches an element in a list	3.2	23.7	3.6	37.1
insert inserts an element into a sorted list	5.1	50.1	5.4	46.8
delete removes an element from a sorted list	3.7	41.7	3.9	35.8
append appends two lists	3.7	18.4	3.9	22.5
reverse destructive list-reversal	3.6	26.9	3.4	21.0
revApp reverses a list by appending its head to its reversed tail	4.3	43.6	4.3	41.7
merge merges two sorted lists	12.5	585.1	5.4	87.1
splice splices two lists	4.9	76.5	4.8	33.6
running the running example	5.2	80.5	5.0	36.5
b. Tree manipulating programs	**Space**	**Time**	**Space**	**Time**
create creates a full tree	-	-	2.6	14.3
insert inserts a node	5.4	98.1	5.6	49.6
remove removes a node using `removeRoot` and `spliceLeft`	9.6	480.3	6.6	167.5
find finds a node with a given key	4.9	53.4	6.5	105.7
height returns the tree's height	-	-	5.4	76.1
spliceLeft a tree as the leftmost child of another tree	5.3	51.6	5.3	35.7
removeRoot removes the root of a tree	6.1	107.8	6.1	73.9
rotate rotates the left and right children of every node	-	-	4.9	57.1
c. Sorting programs	**Space**	**Time**	**Space**	**Time**
IinsertionSort moves the list elements into a sorted list	8.6	449.8	7.3	392.2
TailSort inserts the list head to its (recursively) sorted tail	4.9	101.6	4.9	103.4
QuickSort quicksorts a list	-	-	13.5	1017.1

d. [34] (Call String) vs. [19] (Relational) vs. our method						
Method	**Call String**		**Relational**		**Our method**	
Procedure	**Space**	**Time**	**Space**	**Time**	**Space**	**Time**
insert	1.8	20.8	6.3	122.9	3.5	20.0
delete	1.7	16.4	6.8	145.7	2.8	14.9
reverse	1.8	13.9	4.0	6.4	2.8	7.5
reverse8	2.7	123.8	9.1	14.8	2.8	21.7

e. Inline vs. Procedural Abstraction				
	Inline		**Proc. Call**	
Program	**Space**	**Time**	**Space**	**Time**
crt1x3	2.5	5.1	2.5	6.0
crt2x3	4.5	12.5	2.8	7.3
crt3x3	6.4	22.6	3.1	8.6
crt4x3	8.1	38.6	3.3	9.9
crt8x3	17.3	133.4	4.0	15.6

Procedure `reverse8` reverses the same list 8 times. The cost of its analysis indicates that our approach, as well as [19], profits from being able to reuse the summary of `reverse`, while [34] cannot.

In addition, we examined whether our analysis benefits from reuse of procedure summaries. Tab. 4.e shows the cost of the analysis of programs that allocate several lists. Program **crtYx3** allocates Y lists. The table compares the cost of the analysis of programs that allocate a list by invoking `create3` (right column) to that of programs that inline `create3`'s body. The results are encouraging as they indicate (at least in these simple examples) that our analysis benefits from procedural abstraction.

5 Related Work

Interprocedural shape analysis has been studied in [34, 19, 7, 33, 15].

[34] explicitly represents the runtime stack and abstracts it as a linked-list. In this approach, the entire heap, and the runtime stack are represented at every program point. As a result, the abstraction may lose information about properties of the heap, *for parts of the heap that cannot be affected by the procedure at all.*

[19] considers procedures as transformers from the (entire) heap before the call, to the (entire) heap after the call. Irrelevant objects are summarized into a single summary node. Relevant objects are summarized using a two-store vocabulary. One vocabulary records the current properties of the object. The other vocabulary encodes the properties that the object had when the procedure was invoked. The latter vocabulary allows to match objects at the call-site and at the exit-site. Note that this scheme never summarizes together objects that were not summarized together when the procedure was invoked. For cutpoint-free programs, these may lead to needlessly large summaries. Consider for example a procedure that operates on several lists and nondeterministically replaces elements between the list tails. The method of [19] will not summarize list elements that originated from different input lists. Thus, it will generate exponentially more mappings in the procedure summary, than the ones produced by our method.

[33] presents a heap-modular interprocedural shape-analysis for programs manipulating singly linked lists (without implementation). The algorithm explicitly records cutpoint objects in the local heap, and may become imprecise when there is more than one cutpoint. Our algorithm can be seen as a specialization of [33] for handling cutpoint-free programs and as its generalization for handling trees and sorting programs. In addition, because we restricted our attention to cutpoint-free programs, our semantics and analysis are much simpler than the ones in [33].

[15] exploits a staged analysis to obtain a relatively scalable interprocedural shape analysis. This approach uses a scalable imprecise pointer-analysis to decompose the heap into a collection of independent locations. The precision of this approach might be limited as it relies on pointer-expressions appearing in the program's text. Its tabulation operates on global heaps, potentially leading to a low reuse of procedure summaries.

For the special case of singly-linked lists, another approach for modular shape analysis is presented in [7] without an implementation. The main idea there is to record for every object both its current properties and the properties it had at that time the procedure was invoked.

A heap modular interprocedural may-alias analysis is given in [12]. The key observation there is that a procedure operates uniformly on all aliasing relationships involving variables of pending calls. This method applies to programs with cutpoints. However, the lack of *must*-alias information may lead to a loss of precision in the analysis of destructive updates. For more details on the relation between [12] and local-heap shape analysis see [32, Sec. 5.1].

Local reasoning [18, 31] provides a way of proving properties of a procedure independent of its calling contexts by using the "frame rule". In some sense, the approach used in this paper is in the spirit of local reasoning. Our semantics resembles the frame rule in the sense that the effect of a procedure call on a large heap can be obtained from its effect on a subheap. Local reasoning allows for an arbitrary partitioning of the heap

based on user-supplied specifications. In contrast, in our work, the partitioning of the heap is built into the concrete semantics, and abstract interpretation is used to establish properties in the absence of user-supplied specifications.

Another relevant body of work is that concerning *encapsulation* (also known as *confinement* or *ownership*) [1,3,4,5,8,9,14,17,21,25,28]. These works allow modular reasoning about heap-manipulating (object-oriented) programs. The common aspect of these works, as described in [27], is that they all place various restrictions on the sharing in the heap while pointers from the stack are generally left unrestricted. In our work, the semantics allows for arbitrary heap sharing within the same procedure, but restricts both the heap sharing and the stack sharing across procedure calls.

6 Conclusions and Future Work

In this paper, we presented an interprocedural shape analysis for cutpoint-free programs. Our analysis is modular in the heap and thus allows reusing the effect of a procedure at different calling contexts. In the future, we plan to utilize liveness analysis to automatically remove false cutpoints.

Acknowledgments. We are grateful for the helpful comments of N. Dor, S. Fink, T. Lev-Ami, R. Manevich, R. Shaham, G. Yorsh, and the anonymous referees.

References

1. P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *European Conference on Object-Oriented Programming (ESOP)*, 1997.
2. T. Ball and S.K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.
3. A. Banerjee and D. A. Naumann. Representation independence, confinement, and access control. In *Symp. on Princ. of Prog. Lang. (POPL)*, 2002.
4. B. Bokowski and J. Vitek. Confined types. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1999.
5. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Symp. on Princ. of Prog. Lang. (POPL)*, 2003.
6. D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Conf. on Prog. Lang. Design and Impl. (PLDI)*, 1990.
7. S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *International Static Analysis Symposium (SAS)*, 2003.
8. D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *European Conference on Object-Oriented Programming (ESOP)*, 2001.
9. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1998.
10. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *Formal Descriptions of Programming Concepts, (IFIP WG 2.2, St. Andrews, Canada, August 1977)*, pages 237–277. North-Holland, 1978.
11. M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *Conf. on Prog. Lang. Design and Impl. (PLDI)*, 2002.

12. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Conf. on Prog. Lang. Design and Impl. (PLDI)*, 1994.
13. N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *International Static Analysis Symposium (SAS)*, 2000.
14. C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2001.
15. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Symp. on Princ. of Prog. Lang. (POPL)*, 2005.
16. C. A. R. Hoare. Algorithm 64: Quicksort. *Comm. of the ACM (CACM)*, 4(7):321, 1961.
17. J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1991.
18. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Symp. on Princ. of Prog. Lang. (POPL)*, 2001.
19. B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *International Static Analysis Symposium (SAS)*, 2004.
20. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Int. Conf. on Comp. Construct. (CC)*, 1992.
21. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Conf. on Prog. Lang. Design and Impl. (PLDI)*, 2002.
22. T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Int. Symp. on Software Testing and Analysis (ISSTA)*, 2000.
23. T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *International Static Analysis Symposium (SAS)*, 2000. Available at http://www.math.tau.ac.il/∼ tvla.
24. R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *International Static Analysis Symposium (SAS)*, 2004.
25. P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.
26. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
27. J. Noble, R. Biddle, E. Tempero, A. Potanin, and D. Clarke. Towards a model of encapsulation. In *The First International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, 2003.
28. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *European Conference on Object-Oriented Programming (ESOP)*, 1998.
29. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. on Princ. of Prog. Lang. (POPL)*, 1995.
30. T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *European Symposium on Programming Languages (ESOP)*, 2003.
31. J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Symp. on Logic in Computer Science (LICS)*, 2002.
32. N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. Tech. Rep. 1, AVACS, September 2004. Available at "*http://www.avacs.org*".
33. N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Symp. on Princ. of Prog. Lang. (POPL)*, 2005.
34. N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. In *Int. Conf. on Comp. Construct. (CC)*, 2001.
35. N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. Tech. Rep. 104/05, Tel Aviv Uni., April 2005. Available at "*http://www.math.tau.ac.il/∼maon*".

36. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst. (TOPLAS)*, 24(3):217–298, 2002.
37. R. Shaham, E. Yahav, E.K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *International Static Analysis Symposium (SAS)*, 2003.
38. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

Understanding the Origin of Alarms in ASTRÉE

Xavier Rival

École Normale Supérieure, 45, rue d'Ulm,
75230, Paris cedex 5, France

Abstract. Static analyzers like ASTRÉE are incomplete, hence, may produce false alarms. We propose a framework for the investigation of the alarms produced by ASTRÉE, so as to help classifying them as *true errors* or *false alarms* that are due to the approximation inherent in the static analysis. Our approach is based on the computation of an approximation of a set of traces specified by an initial and a (set of) final state(s). Moreover, we allow for finer analyses to focus on some execution patterns or on some possible inputs. The underlying algorithms were implemented inside ASTRÉE and used successfully to track alarms in large, critical embedded applications.

1 Introduction

The risk of failure due to software bugs is no longer considered acceptable in the case of critical applications (as in aerospace, energy, automotive systems). Therefore, sound program analyzers have been developed in the last few years, that aim at proving safety properties of critical, embedded software such as memory properties [22], absence of runtime errors [3], absence of buffer overruns [11], correctness of pointer operations [28]. These tools attempt to prove automatically the correctness of programs, even though this is not decidable; they are sound (they never claim the property of interest to hold even though it does not) and always terminate; hence, they are *incomplete*: they may produce false alarms, i.e. report not being able to prove the correctness of some critical operation even though no concrete execution of the program fails at this point.

Alarms are a major issue for end-users. Indeed, in case the analyzer reports an alarm, it could either be a false alarm or a real bug that should be fixed. Ideally, a report for a true error should come with an error scenario. Currently, the alarm investigation process in ASTRÉE [3] mainly relies on the manual inspection of invariants, partly with a graphical interface [10]; this process turns out to be cumbersome, since even simple alarms may take days to classify.

A false alarm might either be due to an imprecision of some abstract operations involved in the analysis (e.g. the abstract join operator usually loses precision) as in Fig. 1(a) (simplified version of an alarm formerly reported by ASTRÉE) or to a lack of expressiveness of the domain (checking the example of Fig. 1(c) requires proving a very deep arithmetic theorem). In the former case, we may expect a (semi)-automatic refinement process to prove the alarm to be

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 303–319, 2005.
© Springer-Verlag Berlin Heidelberg 2005

```
l0 : input(x);
l1 : if(x > 0){y = x;}
     else{y = −x;}
l2 : b = (y > 10);
l3 : assert(b ⇒ (x < −10 ∨ 10 < x));
```
(a)

```
l0 : x = 1; y = 1;
     while(true){
li :   input(x);
ld :   assert(y > 0);
       y = x;}
```
(b)

```
x, y, z are integer variables
input(x); input(y); input(z);
if(x > 0 ∧ y > 0 ∧ z > 0){
  assert(x^4 ≠ y^4 + z^4);
}
```
(c)

Fig. 1. Three example programs

false; in the latter, the design of a refined domain can hardly be automated, so we can only hope for a refined alarm description.

Our goal is to provide some support in the alarm investigation process. We propose to resort to automatic, sound static analysis techniques so as to refine an initial static analysis into an approximation of a subset of traces that actually lead to an error. If a combination of forward and backward refining analyses allows to prove that this set is empty, we can conclude the alarm is false (as in Fig. 1(a)); otherwise, we get a refined characterization of the (possibly fictitious) erroneous traces. We propose to refine this kind of *semantic slicing* (i.e. extraction of part of the program traces) by selecting some alarm contexts (e.g. traces leading to an error after two iterations in a loop or traces constrained by some set of inputs). A similar process allows to check an error scenario, by slicing the traces reaching the alarm point in some context specified by an execution pattern and a set of inputs that are supposed to be valid: in case the analysis reveals that such conditions *always* lead to an error, the error scenario is validated (this can be achieved in the example of Fig. 1(b)); it is a counter-example.

The contribution of the paper is both theoretical and practical:

- we propose a framework for alarm inspection, based on backward analysis [6,7], trace partitioning [23], and slicing techniques [29,19];
- we provide encouraging practical results obtained by an implementation inside the ASTRÉE static analyzer [3,10].

Sect. 2 introduces forward-backward analysis-based approximation (semantic slicing) of a set of traces resulting in an error. Semantic slicing refinements are introduced in Sect. 3: restriction to some execution patterns in Sect. 3.1 and to some inputs in Sect. 3.2. Sect. 4 applies syntactic slicing techniques to the reduction of the amount of code to analyze. Sect. 5 presents some practical examples. Sect. 6 concludes and reviews related work.

2 Backward Analysis

2.1 Standard Notations

We restrict to a subset of C for the sake of concision. We let $\mathbb{X}$ (resp. $\mathbb{V}$) denote the set of variables (resp. of values); we write $\mathbb{e}$ (resp. $\mathbb{s}$) for the set of expressions (resp. statements aka programs). Variables and values have an integer, floating-point or boolean type. We consider assignments, conditionals, loops, assertions, and an additional **input**(x) statement, that emulates the reading of a volatile

variable x: this statement replaces the value of x with a random value of the corresponding type. The grammar is given below. The control point before each statement and at the end of each block is associated to a unique label $l \in \mathbb{L}$.

$$\begin{array}{l} e\ (e \in \mathbb{e}) ::= v\ \ (\text{where } v \in \mathbb{V})\ \mid\ x\ \ (\text{where } x \in \mathbb{X})\ \mid\ e \oplus e \\ s\ (s \in \mathbb{s}) ::= x := e\ \ (\text{where } x \in \mathbb{X}, e \in \mathbb{e})\ \mid\ s; s\ \mid\ \mathbf{skip} \\ \qquad\qquad \mid\ \mathbf{input}(x)\ \mid\ \mathbf{assert}(e)\ \mid\ \mathbf{if}(e)\{s\}\mathbf{else}\{s\}\ \mid\ \mathbf{while}(e)\{s\} \end{array}$$

In practice, the subset of C we analyze also includes functions, pointers, composite data-structures, all kinds of definitions, and all control structures. It excludes recursive functions, dynamic memory allocation, and destructive updates.

A state $s \in \mathbb{S}$ is either the error state Ω or a pair (l, ρ) where l is a label and $\rho \in \mathbb{M} = \mathbb{X} \to \mathbb{V}$ is a *memory state* (aka *store*). Note that we assume there are no errors in expressions; hence, the error state Ω can only be reached after a failed assertion. The semantics $[\![s]\!] = \{\langle \sigma_0, \ldots, \sigma_n \rangle \mid \forall i,\ \sigma_i \to \sigma_{i+1}\}$ of a program s is a set of sequences of states (aka traces), such that any two successive states are related by the transition relation $\to$ of the program. The relation $\to$ is defined by local rules, such as the following:

- assert statement $l_0 : \mathbf{assert}(e); l_1$: if $[\![e]\!](\rho) = \mathbf{true}$ ($[\![e]\!](\rho)$ is the result of the evaluation of e in ρ), then $(l_0, \rho) \to (l_1, \rho)$; if $[\![e]\!](\rho) = \mathbf{false}$, then $(l_0, \rho) \to \Omega$.
- assignment $l_0 : x := e; l_1$: $(l_0, \rho) \to (l_1, \rho[x \leftarrow [\![e]\!](\rho)])$ (where $[\![e]\!] \in \mathbb{M} \to \mathbb{V}$);
- input statement $l_0 : \mathbf{input}(x); l_1$: if $v \in \mathbb{V}$ has the same type as x, then $(l_0, \rho) \to (l_1, \rho[x \leftarrow v])$;

2.2 Approximation of Dangerous Traces

Dangerous States. We consider a program $P \in \mathbb{s}$. A state σ is *dangerous* if it is not Ω and may lead to Ω in one transition step: $\sigma \to \Omega$. A *dangerous label* is a label l followed by an assertion statement ($l : \mathbf{assert}(e);$). ASTRÉE over-approximates the set of *reachable* dangerous states; hence, our goal is to start with such an over-approximation and to make a diagnosis whether a set of concrete, dangerous states is actually reachable.

Dangerous Traces. First, we are interested in real executions only, that is traces starting from an *initial* state. We let $\mathcal{I} \subseteq \mathbb{S}$ denote the set of initial states. Then, the set of real executions is $\overrightarrow{T} = \{\langle \sigma_0, \ldots, \sigma_n \rangle \in [\![P]\!] \mid \sigma_0 \in \mathcal{I}\} = \mathbf{lfp}_\emptyset \overrightarrow{F}$ where $\overrightarrow{F} : E \mapsto \{\langle \sigma \rangle \mid \sigma \in \mathcal{I}\} \cup \{\langle \sigma_0, \ldots, \sigma_n, \sigma_{n+1} \rangle \mid \langle \sigma_0, \ldots, \sigma_n \rangle \in E \wedge \sigma_n \to \sigma_{n+1}\}$ constructs execution traces forward, and $\mathbf{lfp}_X F$ is the least fixpoint of F greater than X. Second, we restrict to executions ending in a dangerous state (or at a dangerous label). We let $\mathcal{F} \subseteq \mathbb{S}$ denote the set of final states of interest. The set of executions ending in $\mathcal{F}$ is $\overleftarrow{T} = \{\langle \sigma_0, \ldots, \sigma_n \rangle \in [\![P]\!] \mid \sigma_n \in \mathcal{F}\} = \mathbf{lfp}_\emptyset \overleftarrow{F}$ where $\overleftarrow{F}$ is defined in a similar way as $\overrightarrow{F}$: $\overleftarrow{F} : E \mapsto \{\langle \sigma \rangle \mid \sigma \in \mathcal{F}\} \cup \{\langle \sigma_{-1}, \sigma_0, \ldots, \sigma_n \rangle \mid \langle \sigma_0, \ldots, \sigma_n \rangle \in E \wedge \sigma_{-1} \to \sigma_0\}$.

The set of traces of interest is $T = \overrightarrow{T} \cap \overleftarrow{T} = \mathbf{lfp}_{\emptyset} \overrightarrow{F} \cap \mathbf{lfp}_{\emptyset} \overleftarrow{F}$. It is a subset of all program behaviors $[\![P]\!]$; in this respect, we call T a *semantic slice*.

In the following, $\mathcal{F}$ may represent either $\mathcal{F}_l = \{(l,\rho) \mid \rho \in \mathbb{M}\}$ or $\mathcal{F}_{\mathfrak{D},l} = \{(l,\rho) \mid \rho \in \mathbb{M} \wedge (l,\rho) \to \Omega\}$, unless we specify explicitly; the slice T_l (resp. $T_{\mathfrak{D},l}$) defined by $\mathcal{F}_l$ (resp. $\mathcal{F}_{\mathfrak{D},l}$) collects the executions ending at label l (resp. the executions causing an error at label l). T shall represent either T_l or $T_{\mathfrak{D},l}$.

Example 1. In the code of Fig. 1(a), label l_3 is a dangerous label; the set of dangerous states for the corresponding **assert** is $\mathcal{F}_{\mathfrak{D},l_3} = \{(l_3,\rho) \mid \rho \in \mathbb{M} \wedge \rho(b) = \mathbf{true} \wedge -10 \leq \rho(x) \leq 10\}$. The set of initial states is $\mathcal{I} = \{(l_0,\rho) \mid \rho \in \mathbb{M}\}$. Clearly, this program does not cause any error: if $y > 10$ at l_2, then, either $x > 0$ and $x = y > 10$ or $x \leq 0$ and $x = -y < -10$. Hence, we wish to prove that $T_{\mathfrak{D},l_3} = \emptyset$.

Alarm Inspection. Our goal is to determine whether an alarm is true or not. We may fall in either of the following cases:

Case a) **alarm proved false:** if the static analysis proves that $T_{\mathfrak{D},l} = \emptyset$, then the dangerous states in $\mathcal{F}_{\mathfrak{D},l}$ are not reachable and *the alarm is false;*

Case b) **alarm proved true:** if the static analysis proves that any trace in T_l violates the assert (i.e. all traces reaching l cause an error at this point), then *the alarm is a true error;*

Case c) **undecided case:** obviously, we may not be able to conclude to either of the previous cases; then, either *an error would occur in some cases only* (this case is considered in Sect. 3) or *the lack of conclusion* is due to a lack of expressivity of the abstract domain (the refined analysis of the alarm context should help designing a domain refinement).

Trace Approximation. The approximation of sets of traces is based on an abstraction of sets of stores defined by a domain $(D_n^\sharp, \sqsubseteq)$ and a concretization function $\gamma_n : D_n^\sharp \to \mathcal{P}(\mathbb{M})$ [8]. We assume that $D_n^\sharp$ provides a widening operator ∇, approximate binary lub ($\sqcup$) and glb ($\sqcap$) operators, and a least element $\bot$, with the usual soundness assumptions. The domain for approximating sets of executions is defined by $D^\sharp = \mathbb{L} \to D_n^\sharp$ and $\gamma : (I \in D^\sharp) \mapsto \{\langle (l_0,\rho_0),\ldots,(l_n,\rho_n)\rangle \mid \forall i,\ \rho_i \in \gamma_n(I(l_i))\}$: a set of traces is approximated by local invariants, approximating the sets of stores that can be encountered at any label. We let $\mathbf{lfp}^\sharp$ denote an abstract post-fixpoint operator, derived from ∇: if $F : D \to D$, $F \circ \gamma \subseteq \gamma \circ F^\sharp$ and $X \subseteq \gamma(X^\sharp)$, then $\mathbf{lfp}_X F \subseteq \gamma(\mathbf{lfp}^\sharp_{X^\sharp} F^\sharp)$. The domain $D_n^\sharp$ is supposed to feature sound abstract operations $\overrightarrow{\mathit{assign}} : \mathbb{X} \times \mathbb{e} \times D_n^\sharp \to D_n^\sharp$, $\mathit{guard} : \mathbb{e} \times D_n^\sharp \to D_n^\sharp$, $\mathit{forget} : \mathbb{X} \times D_n^\sharp \to D_n^\sharp$ that soundly mimic the concrete assignment, testing of conditions, and reading of inputs (by forgetting the value of the modified variable). For instance, the soundness of guard boils down to $\forall d \in D_n^\sharp,\ \forall \rho \in \gamma_n(d),\ [\![e]\!](\rho) = \mathbf{true} \Rightarrow \rho \in \gamma_n(\mathit{guard}(e,d))$.

We let $\mathcal{I}^\sharp \in D^\sharp$ be a sound approximation of the traces made of just one initial state, i.e. $\{\langle\sigma\rangle \mid \sigma \in \mathcal{I}\} \subseteq \gamma(\mathcal{I}^\sharp)$; we let $\mathcal{F}^\sharp \in D^\sharp$ be a sound approximation of $\mathcal{F}$ in the same way, where $\mathcal{F}$ may be either $\mathcal{F}_l$ or $\mathcal{F}_{\mathfrak{D},l}$. The purpose of the following subsections is to approximate the semantic slice T.

2.3 Forward Analysis

We consider here the approximation of $\overrightarrow{T}$. It is well-known that a sound abstract interpreter in $D^\sharp$ can be derived from $\overrightarrow{F}$. More precisely, we can define a family of functions $\overrightarrow{\delta}_{l,l'} : D^\sharp_n \to D^\sharp_n$ that compute the effect of any transition at the abstract level. The soundness of $\overrightarrow{\delta}_{l,l'}$ writes $\forall \rho, \rho' \in \mathbb{M},\ \forall d \in D^\sharp_n,\ \rho \in \gamma_n(d) \wedge (l, \rho) \to (l', \rho') \Rightarrow \rho' \in \gamma_n(\overrightarrow{\delta}_{l,l'}(d))$ and is a direct consequence of the soundness of the basic abstract operations. The forward abstract interpreter is: $\overrightarrow{F}^\sharp : D^\sharp \to D^\sharp;\quad I \mapsto \lambda(l \in \mathbb{L}). \sqcup \{\overrightarrow{\delta}_{l,l'}(I(l)) \mid l' \in \mathbb{L}\}$ (this presentation leaves the choice for an iteration strategy; ASTRÉE uses a denotational iteration scheme, so as to keep the need for local invariant storage down). The soundness of the forward abstract interpreter is proved by standard abstract interpretation methods [8].

Theorem 1 (Soundness of the forward abstract interpreter). $T \subseteq \overrightarrow{T} \subseteq \gamma(\mathbb{I}_0)$ *where* $\mathbb{I}_0 = \mathbf{lfp}^\sharp_{\mathcal{I}^\sharp} \overrightarrow{F}^\sharp$.

Example 2 (Ex. 1 continued). A simple non relational analysis yields the invariant $b \in \{\mathbf{true}, \mathbf{false}\},\ y \geq 0$ at point l_3; the assertion is not proved safe.

2.4 The Backward Interpreter

We consider the refinement of the approximation $\mathbb{I}_0$ of $\overrightarrow{T}$ (hence, of T) into a better approximation, by taking into account the second fixpoint $\overleftarrow{T}$.

A straightforward way to do this would be to design a backward interpreter in the same way as we did for $\overrightarrow{F}^\sharp$ and to compute the intersection of both analyses. Yet, this approach would not be effective, mainly because in most cases, the greatest pre-conditions are not very precise, so that we would face a major loss of precision. For instance, in the case of a function call through a pointer dereference $(\star\texttt{f})()$, the flow depends on the value of `f` *before* the call; hence, the called function cannot be determined from the state after the call and the backward analysis of such a statement with no data about the pre-condition would be very imprecise ($\star$`f` could be any function in the program). Examples of similar issues when analyzing assignments are given in Sect. 2.5.

Hence, the refining backward interpreter $\overleftarrow{F}^\sharp_r$ takes two elements of $D^\sharp$ as inputs: an invariant to refine and an invariant to propagate backwards. It is based on a family of backward transfer functions $\overleftarrow{\delta}_{l,l'} : D^\sharp \times D^\sharp \to D^\sharp$ maps a pre-condition to refine and a post-condition into a refined pre-condition, as stated by the soundness condition: $\forall \rho, \rho' \in \mathbb{M}, \forall d, d' \in D^\sharp_n,\ \rho \in \gamma_n(d) \wedge \rho' \in \gamma_n(d') \wedge (l, \rho) \to (l', \rho') \Rightarrow \rho \in \gamma_n(\overleftarrow{\delta}_{l,l'}(d, d'))$ (i.e. d is refined into a stronger pre-condition, by taking into account the post-condition d'). The definition for a very simple $\overleftarrow{\delta}_{l,l'}$ operator is given and discussed below. It is based on a backward abstract assignment operator $\overleftarrow{\mathit{assign}} : \mathbb{X} \times \mathbb{e} \times D^\sharp_n \times D^\sharp_n \to D^\sharp_n$, satisfying a similar soundness condition. The design of this operator is detailed in Sect. 2.5.

- assignment $l_0 : x := e;\ l_1$: $\overleftarrow{\delta}_{l_0,l_1}(d_0, d_1) = \overleftarrow{\mathit{assign}}(x, e, d_0, d_1)$
- conditional $l_0 : \mathbf{if}(e)\{l_1^t : s^t;\ l_2^t\}\mathbf{else}\{l_1^f : s^f;\ l_2^f\};\ l_3$:
  $\overleftarrow{\delta}_{l_0,l_1^t}(d_0, d_1) = d_0 \sqcap d_1 \quad \overleftarrow{\delta}_{l_2^t,l_3}(d_2, d_3) = d_2 \sqcap d_3$
  $\overleftarrow{\delta}_{l_0,l_1^f}(d_0, d_1) = d_0 \sqcap d_1 \quad \overleftarrow{\delta}_{l_2^f,l_3}(d_2, d_3) = d_2 \sqcap d_3$
- loop $l_0 : \mathbf{while}(e)\{l_1 : s;\ l_2\};\ l_3$:
  $\overleftarrow{\delta}_{l_0,l_1}(d_0, d_1) = d_0 \sqcap d_1 \quad \overleftarrow{\delta}_{l_2,l_0}(d_2, d_0) = d_2 \sqcap d_0 \quad \overleftarrow{\delta}_{l_0,l_3}(d_0, d_3) = d_0 \sqcap d_3$
- assertion $l_0 : \mathbf{assert}(e);\ l_1$: $\overleftarrow{\delta}_{l_0,l_1}(d_0, d_1) = d_0 \sqcap d_1$
- input $l_0 : \mathbf{input}(x);\ l_1$: $\overleftarrow{\delta}_{l_0,l_1}(d_0, d_1) = d_0 \sqcap \mathit{forget}(x, d_1)$

It might be desirable to improve the precision by locally refining the computation of $\overleftarrow{\delta}_{l,l'}$. Indeed, if $\overrightarrow{\delta}_{l,l'}$ and $\overleftarrow{\delta}_{l,l'}$ are sound, then so is $\overleftarrow{\delta}^{(n)}_{l,l'} : (d, d') \mapsto d^{(n)}$, where: $d^{(0)} = \overleftarrow{\delta}_{l,l'}(d, d')$ and $\forall n \in \mathbb{N},\ d^{(n+1)} = \overleftarrow{\delta}_{l,l'}(d^{(n)},\ \overrightarrow{\delta}_{l,l'}(d^{(n)}))$. This process is known as local iterations [17] and usually allows to improve the precision of backward abstract operations and condition testings. For instance, in the case of the **if** statement, we may replace $\overleftarrow{\delta}_{l_0,l_1^t}$ with $\overleftarrow{\delta}_{l_0,l_1^t}(d_0, d_1) = \mathit{guard}(e, d_0 \sqcap d_1)$. Our experience proved local iterations not extremely useful, in the presence of a refined abstract domain, able to carry out rather expressive constraints.

The backward analyzer is defined by a function $\overleftarrow{F}^\sharp_r : D^\sharp \times D^\sharp \to D^\sharp;\ \ (I, I') \mapsto \lambda(l \in \mathbb{L}). \sqcup \{\overleftarrow{\delta}_{l,l'}(I(l), I'(l')) \mid l' \in \mathbb{L}\}$ and satisfies the soundness result:

Theorem 2 (Soundness of the backward abstract interpreter). $T \subseteq \gamma(\mathbb{I}_1)$ *where* $\mathbb{I}_1 = \mathbf{lfp}^\sharp_{\mathcal{F}^\sharp \sqcap \mathbb{I}_0}[\lambda(I \in D^\sharp).\overleftarrow{F}^\sharp_r(\mathbb{I}_0, I)]$ *and* $\mathbb{I}_0$ *is the result of the forward analysis (Th. 1).*

2.5 Backward Assignment

The Domain. ASTRÉE uses a reduced product of several domains, including an interval domain, constraints among boolean variables or between boolean and scalar variables. Among the numerical relational domains, we can cite octagons [25] that express relations of the form $\pm x \pm y \leq c$ and specific domains like [13], adapted to the analysis of control command software components. Complex expressions can be abstracted prior to evaluation inside the abstract domain into interval linear forms [24]: given an abstract value $d \in D^\sharp_n$, e is abstracted into $e' = \mathbf{lin}(e, d) = \sum_k I_k \cdot x_k$ ($\cdot$ is a product operator, $\forall k$, I_k is a real interval, x_k a variable), such that $\forall \rho \in \gamma_n(d),\ [\![e]\!](\rho) \in [\![e']\!](\rho)$. Linearization allows complex (e.g. non linear) expressions to be analyzed precisely inside relational domains.

We consider now the definition of $\overleftarrow{\mathit{assign}}(x, e, d_{\text{pre}}, d_{\text{post}})$. Note that we assume that the l-value x is resolved exactly; this is always the case in the subset of C introduced in Sect. 2.1. In practice, may-assign (e.g. in the case of arrays) and assignment of pointer values are also taken into account. In the proofs below, we let $\rho \in \gamma_n(d_{\text{pre}})$; we write $v = [\![e]\!](\rho)$ and we also assume $\rho[x \leftarrow v] \in \gamma_n(d_{\text{post}})$.

Boolean Transfer Function. If x is a boolean variable, we let:

$$\overleftarrow{assign}(x, e, d_{\text{pre}}, d_{\text{post}}) = \begin{cases} \quad guard(e, forget(x, guard(x, d_{\text{post}})) \sqcap d_{\text{pre}}) \\ \sqcup\, guard(\neg e, forget(x, guard(\neg x, d_{\text{post}})) \sqcap d_{\text{pre}}) \end{cases}$$

Indeed, let us assume $v = \textbf{true}$. Then $\rho \in \gamma_n(forget(x, guard(x, d_{\text{post}})))$, due to the hypothesis on $\rho[x \leftarrow \textbf{true}]$. Moreover, $[\![e]\!](\rho) = \textbf{true}$, so $\rho \in \gamma_n(guard(e, forget(x, guard(x, d_{\text{post}}))))$, which shows the soundness of the above definition.

Arithmetic Backward Transfer Function. Let us assume now that x has scalar type, e.g. floating point. We let $\textbf{lin}(e, d) = \sum_k I_k \cdot x_k$ be an interval linear form of e. We consider the derivation of new octagon and interval constraints:

- the octagon domain provides backward assignment and guard abstract transfer functions for interval linear form expressions [25];
- the interval information in d_{pre} is refined as follows: we let $\mathfrak{I}_x^{\text{pre}}$ (resp. $\mathfrak{I}_x^{\text{post}}$) denote the interval information about x in d_{pre} (resp. d_{post}) and we compute a refined interval information $\mathfrak{I}_{x_j}^{\text{ref}}$ for x_j. The soundness of linearization implies that $v \in (\sum_{k \neq j} I_k \cdot \mathfrak{I}_{x_k}^{\text{pre}}) + I_j \cdot \rho(x_j)$; hence, if $0 \notin I_j$, $\rho(x_j) \in (v - (\sum_{k \neq j} I_k \cdot \mathfrak{I}_{x_k}^{\text{pre}}))/I_j \subseteq (\mathfrak{I}_x^{\text{post}} - (\sum_{k \neq j} I_k \cdot \mathfrak{I}_{x_k}^{\text{pre}}))/I_j$, so that the definition $\mathfrak{I}_{x_j}^{\text{ref}} = (\mathfrak{I}_x^{\text{post}} - (\sum_{k \neq j} I_k \cdot \mathfrak{I}_{x_k}^{\text{pre}}))/I_j$ is correct. These refined intervals are computed with a floating point-based approximation of the semantics of linear interval forms defined in terms of real numbers [24].

Note that the linearization should be computed using d_{pre}: using d_{post} would be unsound, since the value of the assigned variable changed; d_{pre} is also most useful to get the interval information *before* the assignment; hence, the first argument of $\overleftarrow{assign}$ is crucial to compute a precise and sound d_{pre}.

Example 3 (Backward assignment for intervals). We consider the assignment $x := y \cdot x + z$, $d_{\text{pre}} = \{x \geq 0,\ y \in [1, 2],\ z \in [1, 2], \ldots\}$, $d_{\text{post}} = \{x \in [3, 4], \ldots\}$. Linearization converts it into $x := [1, 2] \cdot x + z$; the backward assignment refines the range for x into $[0.5, 3]$. Local iteration would not improve the precision.

2.6 Iteration Strategies

Iterative Refining Process. At the concrete level, T could be defined as the intersection of two independent fixpoints. However, at the abstract level, the invariant $\mathbb{I}_1$ obtained after one forward analysis and one backward analysis might be refined by further analyses. For instance, in case the backward analysis reveals that no trace is going through the true branch of a conditional l : $\textbf{if}(e)\{s_t\}\textbf{else}\{s_f\}; l' : s'$, a refining forward analysis from $\mathbb{I}_1$ may refine the local invariants inside s', since the possible imprecision due to the least upper bound at l' no longer occurs. Note that a further backward analysis would likely improve the results inside s_f also.

Therefore, we propose to define a refining forward analysis and to iterate the refining forward-backward process [6,9]. The *refining forward analyzer* $\overrightarrow{F}_r^\sharp$:

$D^\sharp \times D^\sharp \to D^\sharp$ is based on the forward analyzer and refines its first argument as the backward analyzer: $\overrightarrow{F}^\sharp_r : (I, I') \mapsto \lambda(l' \in \mathbb{L}).\sqcup \{\overrightarrow{\delta}_{l,l}(I(l)) \sqcap I(l') \mid l \in \mathbb{L}\}$.

The *refining sequence* $(\mathbb{I}_n)_{n\in\mathbb{N}}$ is defined by:

- $\mathbb{I}_0$ has been defined in Th. 1 by $\mathbb{I}_0 = \mathbf{lfp}^\sharp_{\mathcal{I}^\sharp} \overrightarrow{F}^\sharp$;
- if $n \geq 0$, $\mathbb{I}_{2n+1} = \mathbf{lfp}^\sharp_{\mathcal{F}^\sharp \sqcap \mathbb{I}_{2n}} [\lambda(I \in D^\sharp).\overleftarrow{F}^\sharp_r(\mathbb{I}_{2n}, I)]$ (akin to $\mathbb{I}_1$, see Th. 2);
- if $n \geq 0$, $\mathbb{I}_{2n+2} = \mathbf{lfp}^\sharp_{\mathcal{I}^\sharp \sqcap \mathbb{I}_{2n+1}} [\lambda(I \in D^\sharp).\overrightarrow{F}^\sharp_r(\mathbb{I}_{2n+1}, I)]$.

Theorem 3 (Soundness of the forward-backward refinement). *The above process is sound:* $\forall n \in \mathbb{N},\ T \subseteq \gamma(\mathbb{I}_n)$.

The proof is done by induction; it is similar to [9, Chap. 6]. Note that, given $\mathcal{I}^\sharp$ and $\mathcal{F}^\sharp$, the sequence of refined invariants is obtained *fully automatically.*

Example 4 (Ex. 1 continued: refined analysis). We let $\mathcal{F}^\sharp$ be $x \in [-10, 10] \wedge y \geq 0 \wedge b = \mathbf{true}$; clearly $\mathcal{F}_{\mathfrak{D},l_3} \subseteq \gamma_n(\mathcal{F}^\sharp)$. The table below shows the result of the first two refining iterations, using a non relational abstraction:

label	$\mathbb{I}_0$	$\mathbb{I}_1$	$\mathbb{I}_2$
l_1	$\top$	$\bot$	$\bot$
l_2	$y \geq 0$	$-10 < x < 10 \wedge y \geq 10$	$\bot$
l_3	$y \geq 0 \wedge b \in \{\mathbf{true}, \mathbf{false}\}$	$-10 < x < 10 \wedge y \geq 10 \wedge b = \mathbf{true}$	$\bot$

$T_{\mathfrak{D},l_3} = \emptyset$: the second refining iteration proves the correctness of the program, i.e. the alarm was false (Sect. 2.2, Case a).

Local Iterations. The above refinement process is not optimal from the efficiency point of view. In the case of the **if** statement considered above, it amounts to completing the backward analysis of the *whole* program before doing a new forward analysis so as to refine the invariant at label l. We might want to do *local iterations*, that is carrying out forward and backward local analysis steps in a single iteration phase. In practice, we found that the refinement process done with an expressive abstract domain (like the domain present in ASTRÉE) does not require much local iterations. Carrying out iterative refinements on large blocks of code (e.g. functions) was a more efficient strategy.

Implementation of the Interpreters. The forward analyzer ASTRÉE is written as a function that inputs a statement and an abstract pre-condition and returns an abstract post-condition; it is defined in denotational style, recursively on the syntax. The export of invariants is optional and one may choose the labels local invariants are exported at. The refining forward analyzer is based on the latter; a parameter just forces it to compute greater lower bounds with a previously computed invariant. The backward analyzer is very similar (same layout, same iteration strategy).

3 Specifying Alarm Contexts

Sect. 2 described the forward-backward analysis involved in the approximation of the set of "dangerous traces". Yet, it does not solve the following issues:

- if we analyze backwards a statement $\mathbf{while}(e)\{l : \mathbf{assert}(e); \ldots\}$, the backward interpreter computes a least-fixpoint on the loop; at the end of the process the invariant at l approximates not only the states right before an error occurs but also the states encountered one, two, or many iterations before, which results in a massive loss of precision at l;
- after refinement of the invariants, we may have the intuition that $T_{\mathfrak{D},l} \neq \emptyset$; should that case arise, we would like to envisage and check an "error scenario", which needs to be defined.

Example 5. We consider the example shown on Fig. 1(b) along this section. Clearly, this program may fail: it may read a negative input; at the next iteration, y is negative, which causes the assertion to fail. However, if the input is always positive, it does not fail. Last, note that it will not fail in the first iteration. The attempt to determine the alarm raised after computing $\mathbb{I}_0$ using a simple interval analysis leaves us in the undetermined case (Sect. 2.2, Case c).

3.1 Restriction to an Execution Pattern

We propose to extend the semantic slicing introduced in Sect. 2, by specifying some *execution patterns* in addition to the initial and final states: for instance, we may restrict a slice to the executions that cause an error after at least two iterations of the main loop and distinguish the states encountered in the last two iterations when approximating this slice. In practice, we resort to some kind of trace partitioning technique, that fits in the framework of [23].

Restriction to a Pattern. We extend the syntax of the language presented in Sect. 2.1 with a statement $l_0 : \mathbf{cnt}; l_1$. Tne new semantics should keep track of the order such statements are executed in. We propose to abstract this order.

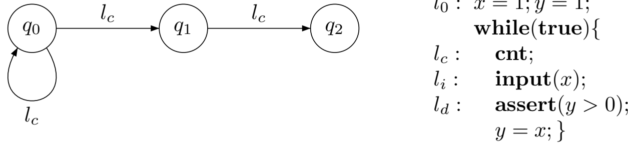
Our approach involves the choice of an *automaton* $(\mathbb{Q}, \rightsquigarrow)$, where $\mathbb{Q}$ is a finite set of states and $(\rightsquigarrow) \subseteq \mathbb{Q} \times \mathbb{L} \times \mathbb{Q}$ is a transition relation (we write $q \overset{l}{\rightsquigarrow} q'$ for $(q, l, q') \in (\rightsquigarrow)$). The labels are replaced with pairs $(l, q) \in \mathbb{L} \times \mathbb{Q}$ in the definition of the concrete semantics: we replace $\mathbb{L}$ with $\mathbb{L}_p = \mathbb{L} \times \mathbb{Q}$; $\mathbb{S}$ with $\mathbb{S}_p = \mathbb{L}_p \times \mathbb{M} \cup \{\Omega\}$. The new, partitioned semantics $[\![P]\!]_p$ is defined similarly to $[\![P]\!]$, using the new transition relation $(\rightarrow_p) \subseteq \mathbb{S}_p \times \mathbb{S}_p$ instead of $(\rightarrow)$:

- case of $l_0 : \mathbf{cnt}$; l_1: if $q_0 \overset{l_0}{\rightsquigarrow} q_1$, then $\forall \rho \in \mathbb{M},\ ((l_0, q_0), \rho) \rightarrow_p ((l_1, q_1), \rho)$;
- case of any other transition:
  if $(l_0, \rho_0) \rightarrow (l_1, \rho_1)$, then $((l_0, q), \rho_0) \rightarrow_p ((l_1, q), \rho_1)$;
  if $(l_0, \rho_0) \rightarrow \Omega$, then $((l_0, q), \rho_0) \rightarrow_p \Omega$.

The *execution pattern* defined by a pair of states (q, q') is $\gamma_{\mathbb{Q}}(q, q') = \{\langle((l_0, q_0), \rho_0), \ldots, ((l_n, q_n), \rho_n)\rangle \mid q_0 = q \wedge q_n = q'\}$.

Example 6 (Ex. 5 continued). We insert a **cnt** statement in the loop and consider the automaton $\mathcal{Q}$ below. Then, $\gamma_{\mathbb{Q}}(q_0, q_2)$ specifies all the traces reaching the dangerous label at the iteration n where $n \geq 2$ and distinguishes the last two iterations (q_1 stands for iteration $n-1$; q_2 stands for iteration n). The automaton

allows us to restrict to the executions that spend more than one iteration in the loop (hence, that may cause an error).



We write $\pi_{\mathbb{L}} : \mathbb{L}_p \to \mathbb{L}$ (resp. $\pi_{\mathbb{S}} : \mathbb{S}_p \to \mathbb{M}, \pi : \mathbb{S}_p^\star \to \mathbb{S}^\star$) for the erasure function that removes the elements of $\mathbb{Q}$ in labels (resp. stores, traces); we let π also be defined for sets of traces. If $\tau \in [\![P]\!]_p$, then $\pi(\tau) \in [\![P]\!]$ (proof obvious).

Refining the Semantic Slice. We also need to extend $\mathcal{I}$ and $\mathcal{F}$. Let $q_i, q_f \in \mathbb{Q}$. We define $\mathcal{I}_p = \{((l, q_i), \rho) \mid (l, \rho) \in \mathcal{I}\}$ and $\mathcal{F}_p = \{((l, q_f), \rho) \mid (l, \rho) \in \mathcal{F}\}$. The automaton $(\mathbb{Q}, \rightsquigarrow)$ and the states q_i, q_f are currently chosen by the user so as to specify some set of execution paths and to specialize even more the semantic slice T; the automatic selection of refinements is left as future work (see discussion in Sect. 5). Other choices for $\mathcal{I}$ or $\mathcal{F}$, involving several states in the automaton are possible (the extension is easy). The definition of T_p from $\mathcal{I}_p, \mathcal{F}_p$ is similar to the definition of T from $\mathcal{I}, \mathcal{F}$ (Sect. 2.2). It satisfies the following property, which clearly shows that it is restricted to the execution pattern defined by q_i, q_f:

$$\pi(T_p) = T \cap \pi(\gamma_{\mathbb{Q}}(q_i, q_f))$$

Approximation of the Semantic Slice. We replace $D^\sharp = \mathbb{L} \to D_n^\sharp$ with the partitioning domain $D_p^\sharp = \mathbb{L} \times \mathbb{Q} \to D_n^\sharp$; we let $\gamma_p : D_p^\sharp \to \mathcal{P}(\mathbb{S}_p), I \mapsto \{\langle((l_0, q_0), \rho_0), \ldots, ((l_n, q_n), \rho_n)\rangle \mid \forall i,\ \rho_i \in \gamma_n(I(l_i, q_i))\}$. The definition of forward and backward abstract interpreters and of the sequence of refined invariants $\mathbb{I}_n^p$ follows the steps of Sect. 2, with extended definitions for $\overrightarrow{\delta}_{l,l'}, \overleftarrow{\delta}_{l,l'}$:

- Case of l_0 : **cnt**; l_1:
  - forward analysis: if $q_0 \overset{l_0}{\rightsquigarrow} q_1$, $\overrightarrow{\delta}_{(l_0,q_0),(l_1,q_1)}(d) = d$;
  - backward analysis: if $q_0 \overset{l_0}{\rightsquigarrow} q_1$, $\overleftarrow{\delta}_{(l_0,q_0),(l_1,q_1)}(d, d') = d \sqcap d'$;
- Case of other statements: $\overrightarrow{\delta}_{(l_0,q),(l_1,q)}$ (resp. $\overleftarrow{\delta}_{(l_0,q),(l_1,q)}$) is defined like $\overrightarrow{\delta}_{l_0,l_1}$ (resp. $\overleftarrow{\delta}_{l_0,l_1}$) in Sect. 2.

Theorem 4 (Soundness). *The static analysis of the partitioned system leads to a sequence of sound invariants:* $\forall n \in \mathbb{N},\ T_p \subseteq \gamma_p(\mathbb{I}_n^p)$.

Proof: with respect to $[\![P]\!]_p$.

Example 7 (Execution patterns). Many patterns can be encoded easily (we assume that the program is of the form **while**$(e)\{l_c : $ **cnt**$; \ldots\}$):

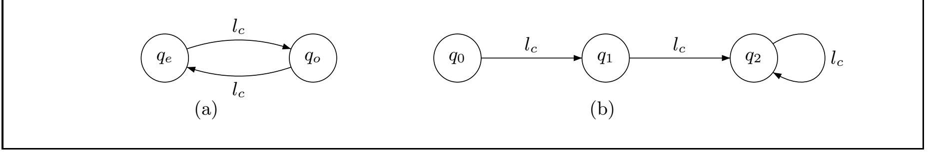


Fig. 2. Automata specifying trace patterns

- On Fig. 2(a), q_e and q_o correspond to even and odd iteration numbers: $\gamma_{\mathbb{Q}}(q_e, q_o)$ slices the traces iterating the loop an even number of times; it also helps distinguishing states reached after an odd (resp. even) number of iterations;
- On Fig. 2(b), $\gamma_{\mathbb{Q}}(q_0, q_1)$ corresponds to the first iteration, $\gamma_{\mathbb{Q}}(q_0, q_2)$ to the n-th iteration ($n \geq 2$).

Example 8 (Ex. 5 continued).

- The refinement of $T_{\mathfrak{D},(l_d,q_1)}$ with the automaton of Fig. 2(b) and the pattern $q_i = q_0, q_f = q_1$ shows that no error may happen in the first iteration;
- Similarly, the refinement of $T_{\mathfrak{D},(l_d,q_2)}$ with the automaton $\mathcal{Q}$ (Ex. 6) gives some intuition about the traces that cause an error: at (l_d, q_2), we get $y \leq 0$; at (l_d, q_1), we get $x \leq 0$, which suggests the input of x should be negative one iteration before the error. We wish now to verify this error scenario.

Remarks. Note that the choice of an automaton with only one state q and such that for any statement l : **cnt**, $q \overset{l}{\rightsquigarrow} q$ results in the same analyses as in Sect. 2.

The trace partitioning presented in this section runs on the top of the one described in [23]; the latter aims at computing more precise invariants thanks to delayed merges of flows (e.g. out of **while** or **if** statements). Our goal here is to extract some execution patterns and to refine the corresponding invariants.

3.2 Restriction to a Set of Inputs

We now consider the slices defined by constraining the inputs; for instance, this may allow to show that this input always leads to an error.

Specification of Inputs. We let $\mathbb{L}_{\text{in}}$ denote the set of **input** statements labels: $\mathbb{L}_{\text{in}} = \{l \in \mathbb{L} \mid l : \textbf{input}(x_l)\}$. An *input specification* is a function $\nu : \mathbb{L}_p \rightarrow \mathcal{P}(\mathbb{V})$, mapping a label to the set of values that may be read at this point. The definition of ν over the partitioned labels allows to select different inputs for different execution contexts (corresponding to different states in the automaton introduced in Sect. 3.1) at the same label. The denotation of the input function ν is the set of traces $\gamma_{\mathbb{V}}(\nu) = \{\langle((l_0, q_0), \rho_0), \ldots, ((l_n, q_n), \rho_n)\rangle \mid \forall i, l_i \in \mathbb{L}_{\text{in}} \Rightarrow \rho_{i+1}(x_{l_i}) \in \nu(l_i, q_i)\}$: such traces satisfy the property that reading an input at label (l, q) yields a value in $\nu(l, q)$.

Refining the Semantic Slice. The semantic slice constrained to ν is:

$$T_v = T_p \cap \gamma_{\mathbb{V}}(\nu) = T \cap \gamma_{\mathbb{Q}}(q_i, q_f) \cap \gamma_{\mathbb{V}}(\nu) \ .$$

Approximation of the Semantic Slice. The only modification required to take into account the input specification concerns the rule for the l_0 : **input**(x); l_1 statement. In this case, we let $\overrightarrow{\delta}_{(l_0,q),(l_1,q)}(d) = \mathit{forget}(x, d) \sqcap \nu^\sharp(l_0, q)$ where $\nu^\sharp(l_0, q)$ is a sound approximation of $\nu(l_0, q)$: $\{\rho \in \mathbb{M} \mid \rho(x) \in \nu(l_0, q)\} \subseteq \gamma_n(\nu^\sharp(l_0))\}$. The case of the backward analysis requires no modification.

Theorem 5 (Soundness). *The resulting abstract interpreters are sound and compute a sequence of sound refined invariants:* $\forall n \in \mathbb{N},\ T_v \subseteq \gamma_p(\mathbb{I}_n^p)$.

The proof follows the definition of a variation $[\![P]\!]_v$ of the concrete semantics $[\![P]\!]_p$: $[\![P]\!]_v$ is obtained from $\rightarrow_{p,\nu}$ just as $[\![P]\!]_p$ from $\rightarrow_p$, where $\rightarrow_{p,\nu}$ is the transition relation constrained by the input function ν. The only modification in the definition of $\rightarrow_{p,\nu}$ comes from the case of the l : **input**(x_l); l' statement: $((l, q), \rho) \rightarrow ((l', q), \rho[x \leftarrow v])$ where $v \in \nu(l, q)$.

Example 9 (Ex. 5 continued). Let us consider the input specification $\nu(l_i, q_1) = -1$. Then, $\mathbb{I}_0$ shows that $y = -1$ at point (l_d, q_2) (the interval analysis proves this property). Hence, the automaton $\mathcal{Q}$ and the input specification ν define a valid error scenario: any execution iterating the loop n times and such that the value read for x during the $(n-1)$th iteration is -1 will result in an error. Such situations are feasible, so the static analysis showed a real bug in the program (Sect. 2.2, Case b).

Example 10 (Ex. 7 continued). The automaton of Fig. 2(a) allows to specify a cyclic input; the automaton of Fig. 2(b) allows to isolate initialization inputs read during the first iteration and inputs read at iteration n for $n \geq 2$.

Currently, the function $\nu^\sharp$ should be provided by the user; further work should allow to synthesize an input specification $\nu^\sharp$ exhibiting an error.

4 Slicing

The size of the program is a major issue when computing semantic slices as suggested in Sect. 2.4 and 2.6. Indeed, forward-backward analyses require saving local invariants, which would induce a dramatic memory cost, if applied to the whole program. Therefore, we propose to use regular, syntactic slicing techniques [29,19] so as to restrict the amount of code to apply the refining analyses to. We use the notations of Sect. 2 for the sake of simplicity (even though mixing slicing techniques and the methods introduced in Sect. 3 is straightforward).

We assume a program s is given, that contains a statement l_d : **assert**(e). We write **use**(e) for the set of variables that appear in expression e.

Slicing. A slicing criterion is a set $\mathcal{C} \subseteq \mathbb{L} \times \mathbb{X}$ of pairs made of a label and a variable ; it specifies a set of variables we wish to observe the value of, at some point. A typical choice is $\mathcal{C} = \{(l_d, x) \mid x \in \mathbf{use}(e)\}$. See eg. [19] for standard slicing algorithms. Slicing is sound in the sense that it does not remove any behavior of the original program for some observation including the dependence closure of the criterion. As a consequence, the static analysis of the slice yields a safe approximation of the semantic slice of the initial program. Beyond the restriction of the size of the code to analyze, an advantage of considering the slice defined by the l_d : **assert**(e) statement is that most of the remaining statements and variables are relevant to the observation at label l_d, i.e. to the alarm under investigation.

Reducing the Size of Slices. If $\mathbb{I}_n(l) = \bot$, then $\mathbb{I}_n$ proves that the statement at label l is not relevant to the semantic slice of interest T; hence, this statement can be safely removed from the slice and its dependences thrown away, which allows to reduce even more the size of the syntactic slice. Such a transformation preserves the soundness and should speed up the computation of $\mathbb{I}_{n+k}$ $(k \geq 1)$.

Approximation of Slices. Slicing should reduce the programs to analyze to a reasonable size; however, even the slices extracted from some **assert** statements may have prohibitive sizes, when extracted from very large programs, with long, cyclic dependence chains. Thus, we propose to do "aggressive slicing" and to approximate any removed statement in a sound manner during the analysis.

For instance, let us consider the forward analysis of a statement $l_0 : x := e; l_1$ (that should be extracted in the slice). As seen in Sect. 2, the forward abstract transfer function for this statement is $\overrightarrow{\delta}_{l_0,l_1} : d \mapsto \overrightarrow{\mathit{assign}}(x, e, d)$. The aggressive slicing of this statement consists in replacing the previous definition of $\overrightarrow{\delta}_{l_0,l_1}$ with the following: $\overrightarrow{\delta}_{l_0,l_1} : d \mapsto \mathit{forget}(x, d)$. Observe that this is sound: $d \mapsto \mathit{forget}(x, d)$ approximates all the concrete transitions defined by the assignment; hence, this new definition for $\overrightarrow{\delta}_{l_0,l_1}$ leads to a sound forward and backward abstract interpreters (the soundness results of Th. 1, Th. 2, and Th. 3 are preserved). Among the possible strategies to reduce the size of "aggressive slices", we can cite the limiting of dependency chains, the restriction to a given subset of variables or the elimination of loop-carried dependences: these approaches lead to an under-approximation $\widehat{\mathcal{C}}$ of the dependences induced by $\mathcal{C}$.

5 Case Studies

A typical alarm investigation session proceeds as follows:

1. do a forward analysis, determine a superset of the possible errors (Th. 1);
2. choose an alarm to investigate; restrict to a slice including the alarm point;
3. define $\mathcal{I}^\sharp, \mathcal{F}^\sharp$, attempt to prove the alarm wrong with forward-backward refinement (Th. 3), otherwise a more precise alarm context slice is found;
4. in case of failure, specialize the alarm context (Sect. 3.1);
5. in case no attempt to get the analyzer to prove $T_{\mathfrak{D},l} = \emptyset$ succeeds, then attempt to prove the alarm true by choosing a set of inputs (Sect. 3.2).

Application to a Family of Programs. We applied our methodology to the alarms raised by Astrée on a series of 3 early development versions of some critical embedded programs (bugs were not unlikely in the development versions).

Size of the C code (lines)	67 500	233 000	412 000
Number of functions	650	1900	2900
Analysis time ($\mathbb{I}_0$) in sec.	1 300	16 200	37 500
Number of alarms	4	1	0
Alarm names	a_1, a_2, a_3, a_4	a_5	-

Slicing (Sect. 4) showed that a_2 (resp. a_4) is a direct consequence of a_1 (resp. a_3); hence, we restricted to the investigation of a_1, a_3 and a_5. The computation of a semantic slice for the corresponding dangerous states on the slices revealed rather informative conditions on the inputs. Specializing some inputs and carrying out a new, forward analysis *allowed to prove the alarms true*, thanks to an input specification as in Ex. 10. The table below provides some data about the process: the number of input constraints is the number of points an input constraint had to be specified for (Sect. 3.2); the number of execution patterns corresponds to the number of automata we considered (Sect. 3.1). The size of the slices (number of lines, functions and variables) involved in the alarms show that a_1, a_3 were rather subtile; a_5 was much simpler. The number of additional constraints generated during the forward-backward refinement is rather difficult to express simply due to the trace partitioning, and to the use of sophisticated numerical domains; we can only mention that it is much higher than the number of variables or of program points. One forward-backward iteration necessitates a reasonable amount of resources for these slices (up to 1 min., 80 Mb).

Alarm	a_1	a_3	a_5
Size of the slice (lines)	1280	4096	244
Number of functions in the slice	29	115	8
Number of variables in the slice including: int, bool, float variables	215 $15, 60, 146$	883 $122, 553, 208$	30 $7, 11, 23$
Execution patterns	2	2	2
Input constraints	4	4	2

The only manual step is the choice of adequate execution patterns and of constraints on inputs, so as to get an error scenario; in all the above cases, these numbers are very low, which shows the amount of work for the user is very reasonable: only 4 inputs had to be chosen in the most complicated case (a_3). However each of these choices had to be made carefully, with respect to complex conditions on bit-fields and arithmetic values. The choices for the execution patterns to examine only required considering very few simple automata (similar to unrolling of loops, akin to Fig. 2(b) and Ex. 6), so that the selection of execution patterns should be easy to automatize.

All alarms found involve intricate floating point computations. For instance, a_5 is due to a mis-use of (interpolated) trigonometric functions, leading to a possibly negative result, causing a square root computation to fail.

Early Experience Conclusions. The use of the system reduced the alarm investigation time to a few hours in the worst case we faced; the refining analyses are fully automatic and default parameters (fixed number of global forward-backward steps, no local iterations) did not have to be twicked too much to give good results. Fully manual inspection of such alarms would have required days of work and would have made the definition of an error scenario much more involved. Moreover, we could successfully classify all alarms, which means that *no false alarm remains.*

6 Conclusion and Related Work

We proposed a framework adapted to alarm inspection. Early experiments are positive about the ability of the system to reduce the burden of tracking the source of alarms in Astrée: overall, *all considered alarms could be classified* (no case similar to Fig. 1(c) left), which is a very positive result.

Some forms of conditioned slicing [21,4] attack a similar problem. However, these methods are essentially based on a purely syntactic process, not only for the extraction but also for the shape of the result (a slice is a subset of the program statements [29]). Slicing has been employed for debugging tasks. Recent advances in this area led to the implementation of conditioned slicing tools like [14], that could be applied to testing and software debugging [18]. However, our system is able to produce *semantic* slices, i.e. to provide global information about a set of executions instead of a mere syntactic subset of the program; this is a major advantage when investigating complex errors. The downside is the use of more sophisticated algorithms; however, syntactic slicing alone would not help significantly the alarm inspection process in Astrée.

The search for counter-examples and automatic refinement has long been a motivation in the model-checking-based systems, such as [5,2,26,27,16]. In particular, the automatic refinement process plays a great role in the determination of the set of predicates (i.e. abstract domain) needed for a precise analysis [1]. Our goal is to bring such methods in static analyzers like Astrée for a different purpose, i.e. to solve the few, subtile alarms, after an already very precise analysis [3] (the construction of the domain requires no internal refinement process).

Forward-backward analysis schemes have been applied, e.g. in [20], to the inference of safety properties. Some static analysis systems have been extended with counter-examples search facilities: [15] relies on random test generation; [12] uses a symbolic under-approximation of erroneous traces and theorem proving. The main difference is that we chose to start with an over-approximation of erroneous traces until conditions on inputs are precise enough so that a counter-example could be found since the search space for counter-examples was huge in our case, due to the size of the programs. For instance, the systematic exploration

of paths as in [12] over length above 1000, with hundreds of variables would not work. Moreover, we allow abstract error scenario to be tested unlike [15,12]: this reduces the amount of input constraints to fix to a minimum. On the other hand, we leave the automatic generation of counter-examples as a future work.

Future work should make the process more automatic for attempting to discover an error scenario, by proposing input sequences and restricting to adapted alarm contexts (which are user provided in Sect. 3.1 and Sect. 3.2). We also plan to make the choice of slices to analyze (Sect. 4) more sensible, by using the result of the initial forward analysis, to choose which part of the invariant to refine.

Acknowledgments. We wish to thank deeply B. Blanchet, P. Cousot, R. Cousot, J. Feret, C. Hymans, L. Mauborgne, A. Miné, and D. Monniaux for comments on early version of this paper and discussions. We are also grateful to M. Sagiv for interesting discussions about related work.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.
2. T. Ball, M. Naik, and S. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL*, 2003.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety Critical Software. In *PLDI*, 2003.
4. G. Canfora, A. Cimitille, and A. D. Lucia. Condition program slicing. *Information and Software Technology; Special issue on Program Slicing*, 1998.
5. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.
6. P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes.* PhD thesis, 1978.
7. P. Cousot. Semantic foundations of program analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.
8. P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
9. P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.
10. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP*, 2005.
11. N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI*, 2003.
12. G. Erez. Generating counter examples for sound abstract interpretation. Master's thesis, 2004.
13. J. Feret. Static analysis of digital filters. In *ESOP*, 2004.
14. C. Fox, S. Danicic, M. Harman, and R. Hierons. ConSIT: A Conditioned Program Slicing System. *Software - Practice and Experience*, 2004.
15. F. Gaucher, E. Jahier, B. Jeannet, and F. Maraninchi. Automatic state reaching for debugging reactive programs. In *AADEBUG*, 2003.

16. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, pages 361–416, 2000.
17. P. Granger. Improving the results of static analyses programs by local decreasing iteration. In *FSTTCS*, 1992.
18. R. Hierons, M. Harman, C. Fox, , L. Ouarbya, and D. Daoudi. Conditioned slicing supports partition testing. *Journal of Software Testing, Verification and Reliability*, 2002.
19. S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing using Program Dependence Graphs. *Programming Languages and Systems*, 1990.
20. B. Jeannet. Dynamic partitioning in linear relation analysis. *Formal Methods in System Design*, 2003.
21. B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 1988.
22. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, 2000.
23. L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *ESOP*, 2005.
24. A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP*, 2004.
25. A. Miné. *Weakly relational numerical abstract domains.* PhD thesis, 2004.
26. G. Pace, N. Halbwachs, and P. Raymond. Counter-example generation in symbolic abstract model-checking. In *6th International Workshop on Formal Methods for Industrial Critical Systems, FMICS*, 2001.
27. A. Podelski. Software model checking with abstraction refinement. In *VMCAI*, 2003.
28. A. Venet and G. Brat. Precise and efficient array bound checking for large embedded c programs. In *PLDI*, 2004.
29. M. Weiser. Program slicing. In *Proceeding of the Fifth International Conference on Software Engineering*, pages 439–449, 1981.

Pair-Sharing Analysis of Object-Oriented Programs

Stefano Secci and Fausto Spoto

Dipartimento di Informatica, Università di Verona, Italy

Abstract. *Pair-sharing analysis* of object-oriented programs determines those pairs of program variables bound at run-time to overlapping data structures. This information is useful for program parallelisation and analysis. We follow a similar construction for logic programming and formalise the property, or abstract domain, Sh of *pair-sharing*. We prove that Sh induces a Galois *insertion* *w.r.t.* the concrete domain of program states. We define a compositional abstract semantics for the static analysis over Sh, and prove it correct.

1 Introduction

Static analysis determines, at compile-time, properties about the run-time behaviour of computer programs, in order to verify, debug and optimise the code. Abstract interpretation [7, 8] is a framework for defining static analyses from the property of interest (the *abstract domain*), and prove their correctness.

In object-oriented languages such as Java, program variables are bound to data structures, stored in a sharable memory, which might hence overlap. Consider for instance the method `clone` in Figure 1 which performs a shallow copy of a `StudentList`. Its Java-like syntax is defined in Section 3. Variables *out* and *ttail* are local to `clone`, and *out* holds its return value. If variables *sl1* and *sl2* have type `StudentList`, an assignment *sl1*:=*sl2*.`clone`() makes them *share* the `Student`s of *sl2*, which become *reachable* from *sl1*. Without the line *out*.`head` :=*this*.`head` in Figure 1, variables *sl1* and *sl2* would not share anymore.

Possible sharing (or, equivalently, definite non-sharing) has many applications. Namely, assume that *sl1* and *sl2* do *not* share. Then

- We can execute the calls *sl1*.`tail`.`clone`(); and *sl2*.`clone`() on different processors with disjoint memories. Hence sharing analysis can be used for *automatic program parallelisation or distribution*;
- An assignment such as *sl1*.`head` := `new Person` does not affect the class of *sl2*.`head`. Hence sharing analysis improves a given *class analysis*, which determines at compile-time the run-time class of the objects bound to the expressions [17];
- If *sl2* is a non-cyclic list then an assignment *sl1*.`tail` :=*sl2* makes *sl1* non-cyclic. This is not necessarily true if *sl1* and *sl2* share: if *sl1* points to a node of *sl2*, the previous assignment builds a cycle. Hence sharing is useful for non-cyclicity analysis.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 320–335, 2005.
© Springer-Verlag Berlin Heidelberg 2005

```
class Object {}
class Person extends Object { int age; }
class Student extends Person {}
class Car extends Object { int cost; }
class StudentList extends Object {
  Student head;    StudentList tail;
  StudentList clone() with ttail:StudentList is {
    out := new StudentList;
    out.head := this.head;
    ttail := this.tail;
    if (ttail = null) then {} else out.tail := ttail.clone()
  }
}
```

Fig. 1. Our running example: a method that performs a shallow copy of a list

In all examples above, alias information [5, 16] is not enough to reach the same conclusions. Namely, to express the sharing of *sl1* and *sl2* (of type `StudentList`) through aliasing, we must check if *sl1* and *sl2* are aliases, or *sl1*.`head` and *sl2*.`head`, or *sl1*.`tail` and *sl2*.`tail`, or *sl1*.`tail`.`head` and *sl2*.`tail`.`head` and so on. Thus sharing cannot be *finitely* computed from aliasing, which is a *special case* of sharing. Nevertheless, sharing is an abstraction of graph-based representations of the memory used by some alias analyses [5, 16]. Graphs are also used in the only sharing analysis for object-oriented programs we are aware of [13]. However, our goal is to follow previous constructions for logic programming [6, 10, 11, 12] and define a more abstract domain Sh for sharing analysis than graphs. Its elements contain the unordered pairs of program variables allowed to share. We prove that a Galois *insertion* exists between Sh and the concrete domain of program states *i.e.,* Sh is not redundant. This is not easy in a strongly-typed language such as Java, compared to untyped logic programming. We provide correct abstract operations over Sh in order to implement a static analysis. We use a denotational semantics, and abstract denotations are mappings over Sh which we can implement through efficient binary decision diagrams [3], by identifying each pair of program variables with a distinct binary variable. Moreover, denotational semantics yields a compositional analysis [18].

We preferred pair-sharing to full sharing [10], which determines the *sets* of variables which share a given data-structure. Our choice is motivated by the fact that abstract domains for pair-sharing should be simpler and smaller than abstract domains for full sharing [1]. There has been some discussion on the redundancy of sharing *w.r.t.* pair-sharing in logic programs [1, 4], whose conclusions, however, do not extend immediately beyond the logic programming realm. In any case, our construction can be easily rephrased for full sharing.

The rest of the paper is organised as follows. Section 2 contains the preliminaries. Section 3 shows our simple language. Section 4 defines the abstract domain Sh and proves the Galois insertion property. Section 5 defines an abstract semantics (analyser) over Sh and states its correctness. Section 6 concludes. Proofs are in [14].

2 Preliminaries

A total (partial) function f is denoted by $\mapsto$ ($\rightarrow$). The *domain* (*codomain*) of f is $\mathrm{dom}(f)$ ($\mathrm{rng}(f)$). We denote by $[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$ the function f where $dom(f) = \{v_1, \ldots, v_n\}$ and $f(v_i) = t_i$ for $i = 1, \ldots, n$. Its *update* is $f[w_1 \mapsto d_1, \ldots, w_m \mapsto d_m]$, where the domain may be enlarged. By $f|_s$ ($f|_{-s}$) we denote the *restriction* of f to $s \subseteq \mathrm{dom}(f)$ (to $\mathrm{dom}(f) \setminus s$). If $f(x) = x$ then x is a *fixpoint* of f. The composition $f \circ g$ of functions f and g is such that $(f \circ g)(x) = g(f(x))$ so that we often denote it as gf. The two components of a *pair* are separated by $\star$. A definition of S such as $S = a \star b$, with a and b meta-variables, silently defines the pair selectors $s.a$ and $s.b$ for $s \in S$.

A *poset* $S \star \leq$ is a set S with a reflexive, transitive and antisymmetric relation $\leq$. If $s \in S$ then $\downarrow s = \{s' \in S \mid s' \leq s\}$. An *upper (respectively, lower) bound* of $S' \subseteq S$ is an element $u \in S$ such that $u' \leq u$ (respectively, $u' \geq u$) for every $u' \in S'$. A *complete lattice* is a poset $C \star \leq$ where *least* upper bounds (*lub*, $\sqcup$) and *greatest* lower bounds (*glb*, $\sqcap$) always exist. If $C \star \leq$ and $A \star \preceq$ are posets, $f : C \mapsto A$ is *(co-)additive* if it preserves lub's (glb's).

Let $C \star \leq$ and $A \star \preceq$ be two posets (the concrete and the abstract domain). A *Galois connection* [7, 8] is a pair of monotonic maps $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ such that $\gamma\alpha$ is extensive and $\alpha\gamma$ is reductive. It is a *Galois insertion* when $\alpha\gamma$ is the identity map *i.e.,* when the abstract domain does not contain *useless* elements. This is equivalent to α being onto, or γ one-to-one. If C and A are complete lattices and α is additive (respectively, γ is co-additive), it is the abstraction map (respectively, concretisation map) of a Galois connection. An abstract operator $\hat{f} : A^n \mapsto A$ is *correct w.r.t.* $f : C^n \rightarrow C$ if $\alpha f \gamma \preceq \hat{f}$.

3 The Language

We describe here our simple Java-like object-oriented language.

Syntax. Variables have a type and contain values. We do not consider primitive types since their values cannot be shared but only copied.

Definition 1. *Each program in the language has a set of* variables *(or* identifiers*)* $\mathcal{V}$ *(including res, out, this) and a finite set of* classes *(or* types*)* $\mathcal{K}$ *ordered by a* subclass relation $\leq$ *such that* $\mathcal{K} \star \leq$ *is a poset. A* type environment *describes a finite set of variables with associated class. It is any element of the set* $TypEnv = \{\tau : \mathcal{V} \rightarrow \mathcal{K} \mid \mathrm{dom}(\tau)$ *is finite*$\}$*. In the following,* τ *will stand for a type environment. Type environments describe the variables in scope in a given program point. Moreover, we write* $F(\kappa)$ *for the type environment that maps the fields of the class* $\kappa \in \mathcal{K}$ *to their type.*

Example 2. In Figure 1, $\mathcal{K} = \{\texttt{Object}, \texttt{Person}, \texttt{Student}, \texttt{Car}, \texttt{StudentList}\}$, where `Object` is the top of the hierarchy and `Student` $\leq$ `Person`. Since we are not interested in primitive types, we have $F(\texttt{Object}) = F(\texttt{Student}) = F(\texttt{Person}) = F(\texttt{Car}) = []$ and $F(\texttt{StudentList}) = [\texttt{head} \mapsto \texttt{Student}, \texttt{tail} \mapsto \texttt{StudentList}]$.

Our expressions and commands are normalised versions of those of Java. For instance, only distinct variables can be the actual parameters of a method call; left-values in assignments can only be a variable or the field of a variable; conditional can only test for equality or nullness of variables; loops must be implemented through recursion. These simplifying assumptions can be relaxed without affecting subsequent results. Instead, it is significant that we allow downwards casts, since our notion of reachability (Definition 11) depends from their presence.

Definition 3. *Our simple language is made of* expressions[1] *and* commands

$$\begin{aligned} exp &::= \texttt{null}\ \kappa \mid \texttt{new}\ \kappa \mid v \mid v.\texttt{f} \mid (\kappa)v \mid v.\texttt{m}(v_1,\ldots,v_n) \\ com &::= v{:=}exp \mid v.\texttt{f} := exp \mid \{com;\cdots;com\} \\ &\mid \texttt{if}\ v = w\ \texttt{then}\ com\ \texttt{else}\ com \mid \texttt{if}\ v = \texttt{null}\ \texttt{then}\ com\ \texttt{else}\ com \end{aligned}$$

where $\kappa \in \mathcal{K}$ and $v, w, v_1, \ldots, v_n \in \mathcal{V}$ are distinct.

Each method $\kappa.\texttt{m}$ is defined *inside class κ with a statement like*

$$\kappa_0\ \texttt{m}(w_1{:}\kappa_1,\ldots,w_n{:}\kappa_n)\ \texttt{with}\ w_{n+1}{:}\kappa_{n+1},\ldots,w_{n+m}{:}\kappa_{n+m}\ \texttt{is}\ com$$

where $w_1,\ldots,w_n,w_{n+1},\ldots,w_{n+m} \in \mathcal{V}$ are distinct and are not res nor this nor out. Their declared types *are $\kappa_1,\ldots,\kappa_n,\kappa_{n+1},\ldots,\kappa_{n+m} \in \mathcal{K}$, respectively. Variables $w_1,\ldots,w_n$ are the* formal parameters *of the method, $w_{n+1},\ldots,w_{n+m}$ are its* local variables*. The method can also use a variable out of type κ_0 which holds its* return value*. We define $body(\kappa.\texttt{m}) = com$, $returnType(\kappa.\texttt{m}) = \kappa_0$, $input(\kappa.\texttt{m}) = [this \mapsto \kappa, w_1 \mapsto \kappa_1,\ldots,w_n \mapsto \kappa_n]$, $output(\kappa.\texttt{m}) = [out \mapsto \kappa_0]$, $locals(\kappa.\texttt{m}) = [w_{n+1} \mapsto \kappa_{n+1},\ldots,w_{n+m} \mapsto w_{n+m}]$ and $scope(\kappa.\texttt{m}) = input(\kappa.\texttt{m}) \cup output(\kappa.\texttt{m}) \cup locals(\kappa.\texttt{m})$.*

Example 4. Consider `StudentList.clone` (just `clone` later) *i.e.,* the method `clone` of `StudentList` in Figure 1. Then $input(\texttt{clone}) = [this \mapsto \texttt{StudentList}]$, $output(\texttt{clone}) = [out \mapsto \texttt{StudentList}]$ and $locals(\texttt{clone}) = [ttail \mapsto \texttt{StudentList}]$.

Our language is strongly typed *i.e.,* expressions *exp* have a static (compile-time) type $type_\tau(exp)$ in τ, consistent with their run-time values (see [14]).

Semantics. We describe here the *state* of the computation and how the language constructs modify it. We use a denotational semantics, hence compositional, in the style of [18]. However, we use a more complex notion of state, to account for dynamically-allocated and sharable data-structures. By using a denotational semantics, our states contain only a single frame, rather than an activation stack of frames. A method call is hence resolved by *plugging* the interpretation of the method (Definition 9) in its calling context. This is standard in denotational semantics and has been used for years in logic programming [2].

A frame binds variables (identifiers) to locations or *null*. A memory binds such locations to objects, which contain a class tag and the frame for their fields.

[1] The `null` constant is decorated with the class κ induced by its context, as in $v{:=}\ \texttt{null}\ \kappa$, where κ is the type of v. This way we avoid introducing a distinguished type for `null`. You can assume this decoration to be provided by the compiler.

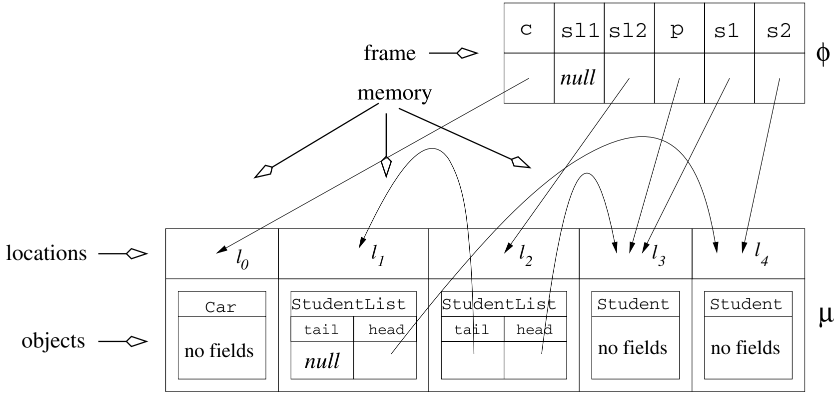


Fig. 2. A state (frame ϕ and memory μ) for $\tau = [c \mapsto \texttt{Car}, \mathit{sl1} \mapsto \texttt{StudentList}, \mathit{sl2} \mapsto \texttt{StudentList}, p \mapsto \texttt{Person}, \mathit{s1} \mapsto \texttt{Student}, \mathit{s2} \mapsto \texttt{Student}]$

Definition 5. *Let Loc be an infinite set of* locations*. We define* frames*,* objects *and* memories *as* $\mathit{Frame}_\tau = \{\phi \mid \phi \in \mathrm{dom}(\tau) \mapsto \mathit{Loc} \cup \{\mathit{null}\}\}$*,* $\mathit{Obj} = \{\kappa \star \phi \mid \kappa \in \mathcal{K},\ \phi \in \mathit{Frame}_{F(\kappa)}\}$ *and* $\mathit{Memory} = \{\mu \in \mathit{Loc} \to \mathit{Obj} \mid \mathrm{dom}(\mu)$ *is finite*$\}$*. A new object of class* κ *is* $\mathit{new}(\kappa) = \kappa \star \phi$*, with* $\phi(v) = \mathit{null}$ *for each* $v \in F(\kappa)$*.*

Example 6. Figure 2 shows a frame ϕ (with 6 variables) and a memory μ. Different occurrences of the same location are linked by arrows. For instance, *s1* is bound to a location l_3 and $\mu(l_3)$ is a `Student` object. Objects are represented as boxes in μ with a class tag and a local frame mapping fields to locations or *null*.

Type correctness $\phi \star \mu : \tau$ guarantees that in ϕ and in the objects in μ there are no dangling pointers and that variables and fields may only be bound to locations which contain objects allowed by τ or by the type environment for the fields of the objects (Definition 1). This is a sensible constraint for the memory allocated by strongly-typed languages, such as Java. For its formal definition, see [14]. We can now define the *states* as type correct pairs $\phi \star \mu$.

Definition 7. *Let* τ *be the type environment at a given program point p. The set of possible* states *at p is* $\Sigma_\tau = \{\phi \star \mu \mid \phi \in \mathit{Frame}_\tau,\ \mu \in \mathit{Memory},\ \phi \star \mu : \tau\}$*.*

Example 8. Consider Figure 2. The variables in ϕ are bound to *null* or to objects of a class allowed by τ. The `tail` fields of the objects in μ are bound to *null* or to a `StudentList`, consistently with $F(\texttt{StudentList})$ (Example 2). The `head` fields are bound to a `Student`, consistently with $F(\texttt{StudentList})$. Hence $\phi \star \mu : \tau$ and $\phi \star \mu \in \Sigma_\tau$.

Each method is *denoted* by a partial function from input to output states. A collection of such functions, one for each method, is an *interpretation*.

Definition 9. *An* interpretation *I maps methods to partial functions on states, such that* $I(\kappa.\mathtt{m}) : \Sigma_{\mathit{input}(\kappa.\mathtt{m})} \to \Sigma_{\mathit{output}(\kappa.\mathtt{m})}$ *for each method* $\kappa.\mathtt{m}$*.*

Definition 10 builds interpretations from the denotations of commands and expressions. These denotations are in [14]. Below, we discuss them informally.

Expressions in our language have side-effects and return a value. Hence their denotations are partial maps from an initial to a final state. The latter contains a distinguished variable *res* holding the value of the expression: $\mathcal{E}^I_\tau[\![_]\!] : exp \mapsto (\Sigma_\tau \to \Sigma_{\tau+exp})$, where $\tau + exp = \tau[res \mapsto type_\tau(exp)]$. Namely, given an input state $\phi \star \mu$, the denotation of `null` κ binds *res* to *null* in ϕ. The denotation of `new` κ binds *res* to a new location bound to a new object of class κ. The denotation of v copies v into *res*. The denotation of $v.$`f` accesses the object $o = \mu(\phi(v))$ bound to v (provided $\phi(v) \neq null$) and then copies the field `f` of o (*i.e.,* $o.\phi(\mathtt{f})$) into *res*. The denotation of $(\kappa)v$ copies v into *res*, but only if the cast is satisfied. The denotation of method call uses the dynamic class of the receiver to fetch the denotation of the method from the current interpretation. It plugs that denotation in the calling context, by building a starting state $\sigma^\dagger$, whose formal parameters (including *this*) are bound to the actual parameters.

The denotation of a command is a partial map from an initial to a final state: $\mathcal{C}^I_\tau[\![_]\!] : com \mapsto (\Sigma_\tau \to \Sigma_\tau)$. Given an initial state $\phi \star \mu$, the denotation of $v{:=}exp$ uses the denotation of *exp* to get a state whose variable *res* holds *exp*'s value. Then it copies *res* into v, and removes *res*. Similarly for $v.\mathtt{f} := exp$, but *res* is copied into the field `f` of the object $\mu\phi(v)$ bound to v, provided $\phi(v) \neq null$. The denotation of the conditionals checks their guard in $\phi \star \mu$ and then uses the denotation of `then` or the denotation of `else`. The denotation of a sequence of commands is the functional composition of their denotations.

By using $\mathcal{C}^I_\tau[\![_]\!]$, we define a transformer on interpretations, which evaluates the bodies of the methods in I, by using an input state where local variables are bound to *null*. At the end, the final state is restricted to the variable *out*, so that Definition 9 is respected. This corresponds to the *immediate consequence operator* used in logic programming [2].

Definition 10. *The following* transformer on interpretations *tranforms an interpretation I into a new interpretation I' such that*

$$\begin{aligned} I'(\kappa.\mathtt{m}) =& (\lambda\phi \star \mu \in \Sigma_{input(\kappa.\mathtt{m})}.\phi[out \mapsto null, w_{n+1} \mapsto null, \ldots, w_{n+m} \mapsto null] \star \mu) \\ & \circ \mathcal{C}^I_{scope(\kappa.\mathtt{m})}[\![body(\kappa.\mathtt{m})]\!] \circ (\lambda\phi \star \mu \in \Sigma_{scope(\kappa.\mathtt{m})}.(\phi|_{out} \star \mu)). \end{aligned}$$

The denotational semantics *of a program is the least fixpoint of this transformer on interpretations.*

4 An Abstract Domain for Pair-Sharing

We formalise here when two variables *share* and define our abstract domain Sh. We need a notion of *reachability* for locations. A location is reachable if it is bound to a variable or to a field of an object stored at a reachable location.

Definition 11. *Let $\phi \star \mu \in \Sigma_\tau$ and $v \in \mathrm{dom}(\tau)$. We define the set of locations* reachable *from v in $\phi \star \mu$ as $L_\tau(\phi \star \mu)(v) = \cup\{L^i_\tau(\phi \star \mu)(v) \mid i \geq 0\}$, where*

$L^0_\tau(\phi\star\mu)(v) = \{\phi(v)\} \cap Loc$ *and* $L^{i+1}_\tau(\phi\star\mu)(v) = \cup\{\mathrm{rng}(\mu(l).\phi) \cap Loc \mid l \in L^i_\tau(\phi\star\mu)(v)\}$*. Two variables* $v_1, v_2 \in \mathrm{dom}(\tau)$ *share in* $\phi\star\mu$ *if there is a location which is reachable from both i.e., if* $L_\tau(\phi\star\mu)(v_1) \cap L_\tau(\phi\star\mu)(v_2) \neq \emptyset$*.*

Note, in Definition 11, that if an object $o = \mu(l)$ is stored in a reachable location l, then also the locations $\mathrm{rng}(\mu(l).\phi) \cap Loc$ of *all* o's fields are reachable. This reflects the fact that we consider a language with (checked) casts (Section 3), which allow all fields of the objects to be accessed in a program.

Example 12. Consider the state $\sigma = \phi\star\mu$ in Figure 2. For every $i \geq 0$ we have

$$
\begin{array}{llll}
L^0_\tau(\sigma)(c) = \{l_0\} & L^{i+1}_\tau(\sigma)(c) = \emptyset & L^i_\tau(\sigma)(sl1) = \emptyset & \\
L^0_\tau(\sigma)(sl2) = \{l_2\} & L^1_\tau(\sigma)(sl2) = \{l_1, l_3\} & L^2_\tau(\sigma)(sl2) = \{l_4\} & L^{i+3}_\tau(\sigma)(sl2) = \emptyset \\
L^0_\tau(\sigma)(p) = \{l_3\} & L^{i+1}_\tau(\sigma)(p) = \emptyset & L^0_\tau(\sigma)(s1) = \{l_3\} & L^{i+1}_\tau(\sigma)(s1) = \emptyset \\
L^0_\tau(\sigma)(s2) = \{l_4\} & L^{i+1}_\tau(\sigma)(s2) = \emptyset. & &
\end{array}
$$

We conclude that $L_\tau(\sigma)(c) = \{l_0\}$, $L_\tau(\sigma)(sl1) = \emptyset$, $L_\tau(\sigma)(sl2) = \{l_1, l_2, l_3, l_4\}$, $L_\tau(\sigma)(p) = \{l_3\}$, $L_\tau(\sigma)(s1) = \{l_3\}$ and $L_\tau(\sigma)(s2) = \{l_4\}$. Hence, in σ, variable *sl2* shares with *sl2*, *p*, *s1*, *s2*; variable *p* does not share with *s2*; *c* shares only with *c*; *sl1* does not share with any variable, not even with itself.

By using reachability, we refine Definition 9 by requiring that a method does not write into the locations L of the input state which are *not* reachable from the formal parameters, nor read them, so that for instance no location in L is reachable from the method's return value. Programming languages such as Java and that of Section 3 satisfy these constraints. They let us prove the correctness of the abstract counterpart of method call that we define later (Figure 3).

Definition 13. *We refine Definition 9 by requiring that if* $I(\kappa.\mathtt{m})(\phi\star\mu) = (\phi'\star\mu')$ *and* $L = \mathrm{dom}(\mu) \setminus (\cup\{L_{input(\kappa.\mathtt{m})}(\phi\star\mu)(v) \mid v \in \mathrm{dom}(input(\kappa.\mathtt{m}))\})$ *then* $\mu|_L = \mu'|_L$*,* $\phi'(out) \notin L$ *and* $\cup\{\mathrm{rng}(\mu'(l)) \cap L \mid l \in \mathrm{dom}(\mu'|_{-L})\} = \emptyset$*.*

As a first attempt, our abstract domain is the powerset of the unordered pairs of variables in $\mathrm{dom}(\tau)$. The concretisation map says that if (v_1, v_2) belongs to an abstract domain element *sh*, then *sh* allows v_1 and v_2 to share.

Definition 14. *Let* $sh \in \wp(\mathrm{dom}(\tau) \times \mathrm{dom}(\tau))$*. We define*

$$
\gamma_\tau(sh) = \left\{ \sigma \in \Sigma_\tau \,\middle|\, \begin{array}{l} \textit{for every } v_1, v_2 \in \mathrm{dom}(\tau) \\ \textit{if } L_\tau(\sigma)(v_1) \cap L_\tau(\sigma)(v_2) \neq \emptyset \textit{ then } (v_1, v_2) \in sh \end{array} \right\}.
$$

It must be observed, however, that two variables might *never* be able to share if their static types do not let them be bound to overlapping data structures.

Example 15. In the state in Figure 2, variable c does not share with any of the other variables (Example 12). This is not specific to that state. There is no state in Σ_τ where c shares with anything but itself. This is because (Figure 1) a `Car` is not a `Person` nor a `Student` nor a `StudentList` nor vice versa. Moreover, it is not possible to reach a shared object from a `Car` and a `Person` (or a `Student` or a `StudentList`) because these classes have no field of the same type.

Example 15 must be taken into account if we are looking for a Galois *insertion*, rather than a Galois *connection*, between $\wp(\Sigma_\tau)$ and the abstract domain. The abstract domain must include only pairs of variables whose static types *share*. As in Definition 11, we first need a notion of *reachability* for classes.

Definition 16. *The set of classes* reachable *in τ from a variable v is $C_\tau(v) = \cup\{C^i_\tau(v) \mid i \geq 0\}$, where $C^0_\tau(v) = \downarrow\tau(v)$ and $C^{i+1}_\tau(v) = \downarrow(\cup\{\mathrm{rng}(F(\kappa)) \mid \kappa \in C^i_\tau(v)\})$. The set of pairs of variables in τ whose static types* share *is*

$$\mathcal{SV}_\tau = \{(v_1, v_2) \in \mathrm{dom}(\tau) \times \mathrm{dom}(\tau) \mid C_\tau(v_1) \cap C_\tau(v_2) \neq \emptyset\}.$$

In Definition 16, if a class κ is reachable, then all its subclasses $\downarrow\kappa$ are considered reachable. This reflects the fact that we consider a language with (checked) casts.

Example 17. Consider τ as in Figure 2. For every $i \geq 0$ we have $C^0_\tau(c) = \{\texttt{Car}\}$, $C^{i+1}_\tau(c) = \emptyset$, $C^0_\tau(sl1) = \{\texttt{StudentList}\}$, $C^{i+1}_\tau(sl1) = \{\texttt{StudentList}, \texttt{Student}\}$, $C^0_\tau(sl2) = \{\texttt{StudentList}\}$, $C^{i+1}_\tau(sl2) = \{\texttt{StudentList}, \texttt{Student}\}$, $C^0_\tau(p) = \{\texttt{Student}, \texttt{Person}\}$, $C^{i+1}_\tau(p) = \emptyset$, $C^0_\tau(s1) = \{\texttt{Student}\}$, $C^{i+1}_\tau(s1) = \emptyset$, $C^0_\tau(s2) = \{\texttt{Student}\}$ and $C^{i+1}_\tau(s2) = \emptyset$. Hence $C_\tau(c) = \{\texttt{Car}\}$, $C_\tau(sl1) = C_\tau(sl2) = \{\texttt{StudentList}, \texttt{Student}\}$, $C_\tau(p) = \{\texttt{Student}, \texttt{Person}\}$ and $C_\tau(s1) = C_\tau(s2) = \{\texttt{Student}\}$. So $\mathcal{SV}_\tau = (\mathrm{dom}(\tau) \times \mathrm{dom}(\tau)) \setminus \{(c, sl1), (c, sl2), (c, p), (c, s1), (c, s2)\}$ *i.e.,* c can only share with c; all other variables can share with each other.

Abstract domain elements should only include pairs in $\mathcal{SV}_\tau$, since the others cannot share. A further observation shows that if v_1 and v_2 share, then they are not *null*. Thus v_1 shares with v_1 and v_2 shares with v_2. Also this constraint is needed to prove the Galois insertion property (Proposition 20).

Definition 18. *The* abstract domain for pair-sharing *is*

$$\mathsf{Sh}_\tau = \{sh \subseteq \mathcal{SV}_\tau \mid \textit{if } (v_1, v_2) \in sh \textit{ then } (v_1, v_1) \in sh \textit{ and } (v_2, v_2) \in sh\}$$

ordered by set-inclusion. From now on, by γ_τ we mean the restriction to Sh_τ of the map γ_τ of Definition 14.

Example 19. Let τ be as in Figure 2. Then $sh_1 = \{(c, sl1), (c, c), (sl1, sl1)\} \notin \mathsf{Sh}_\tau$ since $(c, sl1) \notin \mathcal{SV}_\tau$ (Example 17); $sh_2 = \{(sl1, sl2), (sl1, sl1)\} \notin \mathsf{Sh}_\tau$ since $(sl1, sl2) \in sh_2$ but $(sl2, sl2) \notin sh_2$; $sh_3 = \{(sl1, sl2), (sl1, sl1), (sl2, sl2)\} \in \mathsf{Sh}_\tau$.

Proposition 20. *The map γ_τ of Definition 18 is the concretisation map of a Galois insertion from $\wp(\Sigma_\tau)$ to Sh_τ.*

In a Galois insertion, the concretisation map induces the abstraction map. Its explicit definition, below, states that the abstraction of a set of concrete states S is the set of pairs of variables which share in at least one $\sigma \in S$.

Proposition 21. *The abstraction map induced by the concretisation map of Definition 14 (restricted to Sh_τ) is such that, for every $S \subseteq \Sigma_\tau$,*

$$\alpha_\tau(S) = \left\{ (v_1, v_2) \in \mathrm{dom}(\tau) \times \mathrm{dom}(\tau) \;\middle|\; \begin{array}{l} \textit{there exists } \sigma \in S \textit{ such that} \\ L_\tau(\sigma)(v_1) \cap L_\tau(\sigma)(v_2) \neq \emptyset \end{array} \right\}.$$

Example 22. Consider the state $\phi \star \mu$ in Figure 2. Its reachability information is given in Example 12 so that (remember that pairs are unordered) $\alpha_\tau(\{\phi \star \mu\}) = \{(c, c), (\mathit{sl2}, \mathit{sl2}), (\mathit{sl2}, p), (\mathit{sl2}, \mathit{s1}), (\mathit{sl2}, \mathit{s2}), (p, p), (p, \mathit{s1}), (\mathit{s1}, \mathit{s1}), (\mathit{s2}, \mathit{s2})\}$.

5 An Abstract Semantics on Sh.

The domain Sh_τ of Section 4 induces an abstract version of the semantics of Section 3, which we make explicit here. This semantics is an actual static analyser for pair-sharing which can be implemented inside generic engines such as our Julia analyser [15].

We start with the abstract counterpart of the interpretations of Definition 9. The idea is to map the approximation over Sh_τ of some input states into the approximation of the corresponding output states.

Definition 23. *A* sharing interpretation I *maps methods into total functions such that* $I(\kappa.\mathtt{m}) : \mathsf{Sh}_{input(\kappa.\mathtt{m})} \mapsto \mathsf{Sh}_{output(\kappa.\mathtt{m})}$ *for each method* $\kappa.\mathtt{m}$.

Example 24. Consider the method `clone` in Figure 1. We have $\mathsf{Sh}_{input(\mathtt{clone})} = \{\emptyset, \{(\mathit{this}, \mathit{this})\}\}$ and $\mathsf{Sh}_{output(\mathtt{clone})} = \{\emptyset, \{(\mathit{out}, \mathit{out})\}\}$. A sharing interpretation, consistent with the concrete semantics of the method, is $I = [\emptyset \mapsto \emptyset, \{(\mathit{this}, \mathit{this})\} \mapsto \{(\mathit{out}, \mathit{out})\}]$ *i.e.,* in the input, *this* shares with *this* if and only if, in the output, *out* shares with *out*.

Our goal now is to compute the interpretation of Example 24 automatically.

5.1 Abstract Denotation for the Expressions

The concrete semantics of Section 3 specifies how each expression *exp* transforms an initial state into a final state, where *res* holds the value of *exp*. To mimic this behaviour on the abstract domain, we specify how *exp* transforms input abstract states *sh* into final abstract states sh' where *res* refers to *exp*'s value. For correctness (Section 2), sh' must include the pairs of variables which share in the concrete states σ' obtained by evaluating *exp* from a concrete state $\sigma \in \gamma_\tau(sh)$.

The concrete semantics of $null\ \kappa$ stores *null* in the variable *res* of σ', which otherwise coincides with σ. Hence, in σ', variable *res* does not share. The other variables share exactly as they do in σ. Consequently, we let $sh' = sh$.

The concrete semantics of $\mathtt{new}\ \kappa$ stores in *res* a reference to a new object o, whose fields are *null*. The other variables do not change. Since o is only reachable from *res*, variable *res* shares with itself only. Then we let $sh' = sh \cup \{(res, res)\}$.

The concrete semantics of v obtains σ' from σ by copying v into *res*. Hence, in σ', variable *res* shares with v and all those variable that v used to share with in σ. Since the other variables are unchanged, we let $sh' = sh \cup (sh[v \mapsto res]) \cup \{(v, res)\}$. By $sh[v \mapsto res]$ we mean *sh* where v is renamed into *res*. We improve this approximation for the case when $(v, v) \notin sh$ *i.e.,* when v is definitely *null* so that variable v does not occur in *sh* (Definition 18) and $sh[v \mapsto res] = sh$.

Moreover, in such a case, v and *res* are *null* in σ' and do not share. Hence, in this case we let $sh' = sh$.

When it is defined, the cast $(\kappa)v$ stores in *res* the value of v. Hence the above approximation for v is also correct for $(\kappa)v$.

The concrete semantics of v.f stores in *res* the value of the field f of v, provided v is not *null*. When $(v, v) \notin sh$, variable v is *null* in σ, v.f *never* yields a final state and the best approximation of the resulting, empty set of final states is $\emptyset$. If instead $(v, v) \in sh$, variable *res* shares in σ' with a variable, say w, only if v shares in σ with w: from v one reaches v.f which is an alias of *res*. Moreover, v and *res* share in σ'. Thus we should let $sh' = sh \cup (sh[v \mapsto res]) \cup \{(v, res)\}$. However, Example 25 shows that sh' might contain pairs not in $\mathcal{SV}$ (Definition 16) and hence in general $sh' \notin \mathsf{Sh}$ (Definition 18).

Example 25. Let every `Student` be paired with its `Car` in the list:

```
class StudentCarList extends StudentList { Car car; }
```

Let $\tau = [v \mapsto \texttt{StudentCarList}, w \mapsto \texttt{Student}]$ and $sh = \{(v, w), (v, v), (w, w)\} \in \mathsf{Sh}_\tau$, so that $(res, w) \in sh'$. But a `Car` cannot share with a `Student` (Figure 1) *i.e.,* $(res, w) \notin \mathcal{SV}_{\tau+v.\texttt{car}}$ and $sh' \notin \mathsf{Sh}_{\tau+v.\texttt{car}}$.

We solve this problem by removing spurious pairs such as (res, w) in Example 25. Namely, we define $sh' = sh \cup [(sh[v \mapsto res]) \cap \mathcal{SV}_{\tau+v.\texttt{f}}] \cup \{(v, res)\}$.

The concrete semantics of the method call $v.\texttt{m}(v_1, \ldots, v_n)$ builds an input state $\sigma^\dagger = [this \mapsto \phi(v), w_1 \mapsto \phi(v_1), \ldots, w_n \mapsto \phi(v_n)] \star \mu$ for the callee *i.e.,* it restricts ϕ to $pars = \{v, v_1, \ldots, v_n\}$ and renames v into *this* and each v_i into w_i. We mimic this by restriction and renaming on the abstract domain.

Definition 26. *Let* $sh \in \mathsf{Sh}_\tau$ *and* $V \subseteq \mathrm{dom}(\tau)$. *We define* $sh|_V \in \mathsf{Sh}_\tau$ *as* $sh|_V = \{(v_1, v_2) \in sh \mid v_1 \in V \text{ and } v_2 \in V\}$. *Moreover, we define* $sh|_{-V} = sh|_{\mathrm{dom}(\tau)\setminus V}$.

Let hence $sh^\dagger = sh|_{pars}[v \mapsto this, v_1 \mapsto w_1, \ldots, v_n \mapsto w_n]$ approximate $\sigma^\dagger$. The abstract domain contains no information on the run-time class of v. Hence we conservatively assume that every method m in a subclass of the static type of v might be called [9] *i.e.,* we use $sh^\ddagger = \cup\{I(\kappa.\texttt{m})(sh^\dagger) \mid \kappa \leq \tau(v)\}[out \mapsto res]$ as an approximation for the result of the call. We rename *out* into *res* since, from the point of view of the caller, the returned value of the callee (*out*) is the value of the method call expression (*res*).

We must determine the effects of the call on the variables of the caller. We do it here in a relatively imprecise way. Subsection 5.4 shows how to improve this approximation. We use the fact that a method call can only modify (and access) input locations which are reachable from the actual arguments (Definition 13). Hence we let *res* share with *every* parameter which was not *null* at call-time. Formally, we build the approximation $sh^\flat = sh^\ddagger \cup \{(res, p) \mid (res, res) \in sh^\ddagger,\ p \in pars \text{ and } (p, p) \in sh\}$. Then we close transitively the sharing pairs *w.r.t.* the parameters, by computing the *star-closure* $(sh \cup sh^\flat)^*_{pars}$.

Definition 27. *Let* $sh \in \mathsf{Sh}_\tau$ *and* $V \subseteq \mathrm{dom}(\tau)$. *The* star-closure *of* sh *w.r.t.* V *is* $sh^*_V = sh \cup \big(\{(v_1, v_2) \mid v', v'' \in V,\ (v_1, v') \in sh \text{ and } (v_2, v'') \in sh\} \cap \mathcal{SV}_\tau\big)$.

In Definition 27 we use $\mathcal{SV}_\tau$ to discard pairs of variables which cannot share.

$$\mathcal{SE}^I_\tau[\![\texttt{null}\ \kappa]\!](sh) = sh \qquad \mathcal{SE}^I_\tau[\![\texttt{new}\ \kappa]\!](sh) = sh \cup \{(res, res)\}$$

$$\mathcal{SE}^I_\tau[\![v]\!](sh) = \mathcal{SE}^I_\tau[\![(\kappa)v]\!](sh) = \begin{cases} sh \cup (sh[v \mapsto res]) \cup \{(v, res)\} & \text{if } (v,v) \in sh \\ sh & \text{otherwise} \end{cases}$$

$$\mathcal{SE}^I_\tau[\![v.\texttt{f}]\!](sh) = \begin{cases} sh \cup \{(v, res)\} \cup (sh[v \mapsto res] \cap \mathcal{SV}_{\tau+v.\texttt{f}}) & \text{if } (v,v) \in sh \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathcal{SE}^I_\tau[\![v.\texttt{m}(v_1,\ldots,v_n)]\!](sh) = \begin{cases} (sh \cup sh^\flat)^*_{pars} & \text{if } (v,v) \in sh \\ \emptyset & \text{otherwise} \end{cases}$$

where $pars = \{v, v_1, \ldots, v_n\}$, $sh^\dagger = sh|_{pars}[v \mapsto this, v_1 \mapsto w_1, \ldots, v_n \mapsto w_n]$, $sh^\ddagger = \cup\{I(\kappa.\texttt{m})(sh^\dagger) \mid \kappa \le \tau(v)\}[out \mapsto res]$ and $sh^\flat = sh^\ddagger \cup \{(res, p) \mid (res, res) \in sh^\ddagger,\ p \in pars \text{ and } (p,p) \in sh\}$.

Fig. 3. The sharing interpretation for expressions

Definition 28. *Let τ describe the variables in scope and I be a sharing interpretation. Figure 3 defines the* sharing denotation $\mathcal{SE}^I_\tau[\![_]\!] : exp \mapsto (\mathsf{Sh}_\tau \mapsto \mathsf{Sh}_{\tau+exp})$.

Example 29. Let $\tau = scope(\texttt{clone}) = [out \mapsto \texttt{StudentList}, this \mapsto \texttt{StudentList}, ttail \mapsto \texttt{StudentList}]$ describe the variables in scope in the `clone` method of Figure 1. Let I be the sharing interpretation of Example 24. Then

$$\mathcal{SE}^I_\tau[\![\texttt{new StudentList}]\!](\{(this, this)\}) = \{(res, res), (this, this)\}$$

$$\mathcal{SE}^I_\tau[\![this.\texttt{head}]\!]\left(\left\{\begin{matrix}(out, out),\\(this, this)\end{matrix}\right\}\right) = \left\{\begin{matrix}(out, out), (res, res),\\(this, res), (this, this)\end{matrix}\right\}$$

$$\mathcal{SE}^I_\tau[\![this.\texttt{tail}]\!]\left(\left\{\begin{matrix}(out, out),\\(this, out),\\(this, this)\end{matrix}\right\}\right) = \left\{\begin{matrix}(out, out), (res, out), (res, res),\\(res, this), (this, out), (this, this)\end{matrix}\right\}.$$

Consider now $sh = \{(out, out), (this, out), (this, this), (ttail, this), (ttail, out), (ttail, ttail)\}$. Let us compute $\mathcal{SE}^I_\tau[\![ttail.\texttt{clone}()]\!](sh)$. We have $pars = \{ttail\}$ and $sh^\dagger = \{(this, this)\}$. If we assume that `clone` is not overridden, then $sh^\ddagger = (I(\texttt{clone})(\{(this, this)\}))[out \mapsto res] = \{(res, res)\}$, $sh^\flat = \{(res, res), (res, ttail)\}$ and $(sh \cup sh^\flat)^*_{\{ttail\}} = (\{(out, out), (this, out), (this, this), (ttail, this), (ttail, out), (ttail, ttail)\} \cup \{(res, res), (res, ttail)\})^*_{\{ttail\}}$. This introduces the pairs (out, res) and $(res, this)$ yielding $\{(out, out), (out, res), (res, res), (res, this), (res, ttail), (this, out), (this, this), (ttail, out), (ttail, this), (ttail, ttail)\}$.

5.2 Abstract Denotation for the Commands

In the concrete semantics, each command c transforms an initial state into a final state. On the abstract domain, it transforms an initial state sh into an abstract state sh' which, for correctness (Section 2), includes the pairs of variables which share in the concrete states σ' obtained by evaluating c from each $\sigma \in \gamma_\tau(sh)$.

$$\mathcal{SC}^I_\tau[\![v{:=}exp]\!] = \mathcal{SE}^I_\tau[\![exp]\!] \circ setVar^v_{\tau+exp}$$
$$\text{where } setVar^v_\tau = \lambda sh \in \mathsf{Sh}_\tau . sh|_{-v}[res \mapsto v]$$
$$\mathcal{SC}^I_\tau[\![v.\mathtt{f} := exp]\!] = \mathcal{SE}^I_\tau[\![exp]\!] \circ setField^{v.\mathtt{f}}_{\tau+exp}$$
$$\text{where } setField^{v.\mathtt{f}}_\tau = \lambda sh \in \mathsf{Sh}_\tau . \begin{cases} (((sh \cup \{(v,res)\})^*_{res})|_{-res})^*_v & \text{if } (v,v) \in sh \\ \emptyset & \text{otherwise} \end{cases}$$
$$\mathcal{SC}^I_\tau \left[\!\!\left[\begin{array}{l} \mathtt{if}\ v = w \\ \mathtt{then}\ com_1 \\ \mathtt{else}\ com_2 \end{array} \right]\!\!\right] (sh) = \mathcal{SC}^I_\tau[\![com_1]\!](sh) \cup \mathcal{SC}^I_\tau[\![com_2]\!](sh)$$
$$\mathcal{SC}^I_\tau \left[\!\!\left[\begin{array}{l} \mathtt{if}\ v = \mathtt{null} \\ \mathtt{then}\ com_1 \\ \mathtt{else}\ com_2 \end{array} \right]\!\!\right] (sh) = \begin{cases} \mathcal{SC}^I_\tau[\![com_1]\!](sh|_{-v}) \cup \mathcal{SC}^I_\tau[\![com_2]\!](sh) & \text{if } (v,v) \in sh \\ \mathcal{SC}^I_\tau[\![com_1]\!](sh|_{-v}) & \text{otherwise} \end{cases}$$
$$\mathcal{SC}^I_\tau[\![\{com_1; \ldots; com_p\}]\!] = (\lambda sh \in \mathsf{Sh}_\tau . sh) \circ \mathcal{SC}^I_\tau[\![com_1]\!] \circ \cdots \circ \mathcal{SC}^I_\tau[\![com_p]\!].$$

The identity map $\lambda sh \in \mathsf{Sh}_\tau . sh$ for the sequence of commands is needed when $p = 0$.

Fig. 4. The sharing interpretation for commands

The concrete evaluation of $v{:=}exp$ evaluates exp and stores its result into v. Thus we define sh' as the functional composition of $\mathcal{SE}^I_\tau[\![exp]\!]$ with the map $setVar^v_\tau(sh) = sh|_{-v}[res \mapsto v]$ which renames res into v (v's original value is lost).

Similarly, for $v.\mathtt{f} := exp$ we use a *setField* map. Its definition has two cases. When $(v,v) \notin sh$, we know that v is *null* and hence there is no final state. The best approximation of the empty set of final states is $\emptyset$. Otherwise, its definition reflects the fact that after assigning exp to $v.\mathtt{f}$, variable v might share with every variable w which shares with the value of exp. This means that we must perform a star-closure *w.r.t.* res (Definition 27) and remove res. Moreover if, before this assignment, a variable v' shares with v, then the assignment might also affect v', so we conservatively assume that v' and w might share. This means that we must compute a star-closure *w.r.t.* v. In conclusion, in this second case we let $setField^{v.\mathtt{f}}_\tau(sh) = (((sh \cup \{(v,res)\})^*_{res})|_{-res})^*_v$.

A correct approximation of the conditionals of Definition 3 considers them non-deterministic, so that their denotation is $\mathcal{SC}^I_\tau[\![com_1]\!] \cup \mathcal{SC}^I_\tau[\![com_2]\!]$. But we can do better. Namely, if $(v,v) \notin sh$ then v is definitely *null* in σ, and the guard $v = \mathtt{null}$ is true. Vice versa, in the `then` branch we can assume that the guard is true. When the guard is $v = \mathtt{null}$, this means that v can be removed from the input approximation sh.

The composition of commands is denoted by functional composition over Sh.

Definition 30. *Let τ describe the variables in scope, I be a sharing interpretation. Figure 4 shows the* sharing denotation for commands $\mathcal{SC}^I_\tau[\![_]\!] : com \mapsto (\mathsf{Sh}_\tau \mapsto \mathsf{Sh}_\tau)$.

Example 31. Let $\tau = scope(\mathtt{clone}) = [out \mapsto \mathtt{StudentList}, this \mapsto \mathtt{StudentList}, ttail \mapsto \mathtt{StudentList}]$ describe the variables in scope in the `clone` method of Figure 1. Let I be the sharing interpretation ofExample 24. We want to com-

pute the abstract state sh_5 at the end of `clone` assuming that we run `clone` from $sh_1 = \{(\mathit{this}, \mathit{this})\}$. We use Definition 30 and we write $\{sh_i\}c\{sh_{i+1}\}$ for $\mathcal{SC}^I_\tau[\![c]\!](sh_i) = sh_{i+1}$ *i.e.,* we decorate each program point p with the abstract approximation at p. For the right-hand side of assignments, we use the denotations that we already computed in Example 29. We have

$$
\begin{gathered}
sh_1 = \{(\mathit{this}, \mathit{this})\} \\
\mathit{out} := \texttt{ new StudentList} \\
sh_2 = \{(\mathit{out}, \mathit{out}), (\mathit{this}, \mathit{this})\} \\
\mathit{out}.\texttt{head } := \mathit{this}.\texttt{head} \\
sh_3 = \{(\mathit{out}, \mathit{out}), (\mathit{this}, \mathit{out}), (\mathit{this}, \mathit{this})\} \\
\mathit{ttail} := \mathit{this}.\texttt{tail} \\
sh_4 = \{(\mathit{out}, \mathit{out}), (\mathit{this}, \mathit{out}), (\mathit{this}, \mathit{this}), (\mathit{ttail}, \mathit{out}), (\mathit{ttail}, \mathit{this}), (\mathit{ttail}, \mathit{ttail})\} \\
\texttt{if } \mathit{ttail} = \texttt{ null then } \{\} \texttt{ else } \mathit{out}.\texttt{tail } := \mathit{ttail}.\texttt{clone}() \\
sh_5 = sh_4.
\end{gathered}
$$

Let us consider in detail how sh_5 is computed from sh_4. Since $(\mathit{ttail}, \mathit{ttail}) \in sh_4$,

$$
\begin{aligned}
sh_5 &= \mathcal{SC}^I_\tau[\![\texttt{if} \ldots \mathit{ttail}.\texttt{clone}()]\!](sh_4) \\
&= \mathcal{SC}^I_\tau[\![\{\}]\!](sh_4|_{-\mathit{ttail}}) \cup \mathcal{SC}^I_\tau[\![\mathit{out}.\texttt{tail } := \mathit{ttail}.\texttt{clone}()]\!](sh_4) \\
&= sh_4|_{-\mathit{ttail}} \cup (\mathit{setField}^{\mathit{out}.\texttt{tail}}_{\tau+\mathit{ttail}.\texttt{clone}()}(\mathcal{SE}^I_\tau[\![\mathit{ttail}.\texttt{clone}()]\!](sh_4))) \\
\text{(Ex. 29)} &= sh_4|_{-\mathit{ttail}} \cup \left(\begin{array}{c} \mathit{setField}^{\mathit{out}.\texttt{tail}}_{\tau+\mathit{ttail}.\texttt{clone}()} \\ \underbrace{\left(\left\{ \begin{array}{c} (\mathit{out}, \mathit{res}), (\mathit{out}, \mathit{out}), (\mathit{res}, \mathit{this}), (\mathit{this}, \mathit{out}), \\ (\mathit{this}, \mathit{this}), (\mathit{ttail}, \mathit{this}), (\mathit{ttail}, \mathit{out}), (\mathit{ttail}, \mathit{ttail}), \\ (\mathit{res}, \mathit{res}), (\mathit{res}, \mathit{ttail}) \end{array} \right\} \right)}_{sh} \end{array} \right) \\
&= sh_4|_{-\mathit{ttail}} \cup (((sh \cup \{(\mathit{out}, \mathit{res})\})^*_{\mathit{res}})|_{-\mathit{res}})^*_{\mathit{out}} \\
&= sh_4|_{-\mathit{ttail}} \cup \left(\left\{ \begin{array}{c} (\mathit{out}, \mathit{out}), (\mathit{this}, \mathit{out}), (\mathit{this}, \mathit{this}), \\ (\mathit{ttail}, \mathit{this}), (\mathit{ttail}, \mathit{out}), (\mathit{ttail}, \mathit{ttail}) \end{array} \right\} \right)^*_{\mathit{out}} \\
&= sh_4|_{-\mathit{ttail}} \cup \left(\left\{ \begin{array}{c} (\mathit{out}, \mathit{out}), (\mathit{this}, \mathit{out}), (\mathit{this}, \mathit{this}), \\ (\mathit{ttail}, \mathit{this}), (\mathit{ttail}, \mathit{out}), (\mathit{ttail}, \mathit{ttail}) \end{array} \right\} \right) = sh_4.
\end{aligned}
$$

The approximation sh_5 in Example 31 lets *out* (`clone`'s return value) share with itself (*i.e.,* it might be non-null), with *this* (`clone` performs a shallow clone of the `StudentList` *this*, by sharing the `Students`) and with *ttail* (because of the recursive call). You cannot drop any single pair from sh_5 without breaking the correctness of the analysis. Instead, $(\mathit{out}, \mathit{ttail})$ is redundant in sh_4. It is there since $\mathit{out}.\texttt{head} := \mathit{this}.\texttt{head}$ makes *out* share with *this* and $\mathit{ttail} := \mathit{this}.\texttt{tail}$ makes *ttail* share with *this* and hence, (too) conservatively, with *out*.

5.3 Correctness

The first result of correctness states that the abstract denotations are correct (Section 2) *w.r.t.* the concrete denotations.

Proposition 32. *The abstract denotations of Definitions 28 and 30 are correct.*

The concrete transformer on interpretations (Definition 10) induces an abstract transformer on sharing interpretations.

Definition 33. *Given a sharing interpretation I, we define a new sharing interpretation I' such that $I'(\kappa.\mathtt{m}) = \mathcal{SC}^I_{scope(\kappa.\mathtt{m})}[\![body(\kappa.\mathtt{m})]\!] \circ (\lambda sh \in \Sigma_{scope(\kappa.\mathtt{m})}.sh|_{out})$. The* sharing denotational semantics *of a program is the least fixpoint of this transformer on sharing interpretations.*

The following result follows from Proposition 32.

Proposition 34. *The transformer on sharing interpretations of Definition 33 is correct w.r.t. that on concrete interpretations of Definition 10. Hence, the sharing denotational semantics is a safe approximation of the denotational semantics.*

Example 35. Let us use, in Definition 33, the denotation of Example 31. We get an interpretation $I' = I$, hence a fixpoint of the transformer of Definition 33. We can actually *construct* I (Example 24) as the limit of a Kleene sequence of approximations, as usual in denotational abstract interpretation [7, 8]. Hence it is the *least* fixpoint *i.e.,* `clone`'s sharing denotational semantics.

5.4 Improving the Precision of Method Calls

The denotation for method calls of Definition 28 can be very imprecise.

Example 36. Let us remove the line out.`head` := $this$.`head` from Figure 1. The method `clone` builds now a `StudentList`, as long as *this*, but whose `Student`*s* are *null*. Hence, at the end of `clone`, variable *out* does not share with *this*. Let us verify if our analysis captures that, by re-executing what we did in Example 31.

$$sh_1 = \{(this, this)\}$$
$$\mathtt{out} := \mathtt{new\ StudentList}$$
$$sh_2 = \{(out, out), (this, this)\}$$
$$\mathtt{ttail} := \mathtt{this.tail}$$
$$sh_3 = \{(out, out), (this, this), (this, ttail), (ttail, ttail)\}$$
$$\mathtt{if\ ttail} = \mathtt{null\ then}\ \{\}\ \mathtt{else\ out.tail} := \mathtt{ttail.clone()}$$
$$sh_4 = \{(out, out), (out, this), (this, this), (out, ttail), (this, ttail), (ttail, ttail)\}.$$

Since $(out, this) \in sh_4$, our analysis is *not* able to guarantee that *this* does not share with the result of `clone`.

In Example 36, the problem is that, in order to approximate the recursive call $ttail$.`clone`(), we use a set $sh^\flat$ (Definition 28) which lets the parameters of the call share with its result, if they are not definitely *null*. In our example, $sh^\flat$ contains the spurious pair $(res, ttail)$, which by star-closure introduces further imprecisions, until $(out, this)$ is put in the approximation.

We can improve the precision of the analysis with explicit information on which actual parameters of a method call share with the return value. Hence we enlarge the set of the variables in the final states of the interpretations (Definitions 9 and 23). While $output(\kappa.\mathtt{m})$ provides information on *out* only (Definition 3), we use $output(\kappa.\mathtt{m}) \cup input'(\kappa.\mathtt{m})$ instead, where $input'(\kappa.\mathtt{m})$ are new local

primed variables holding copies of the actual parameters of κ.m. These variables are never modified, so that at the end they provide information on which actual parameters share with *out*, by renaming primed variables into unprimed ones: $sh^\flat = sh^\ddagger[\boldsymbol{v}' \mapsto \boldsymbol{v}]$ for the primed variables $\boldsymbol{v}'$.

Example 37. Let us re-execute the analysis of Example 36 with a primed variable *this*′. We use an interpretation I such that $I(\texttt{clone})(\{(this, this)\}) = \{(out, out), (this', this')\}$ *i.e.,* at the end of `clone` the actual parameter passed for *this* does not share with the result of the method. We want to verify that this interpretation is a fixpoint of our semantics. We have

$$sh_1 = \{(this, this)\}$$
$$\texttt{this}' := \texttt{this} \quad // \; this' \text{ is initially aliased to } this$$
$$sh_2 = \{(this, this), (this, this'), (this', this')\}$$
$$\texttt{out} := \texttt{new StudentList}$$
$$sh_3 = \{(out, out), (this, this), (this, this'), (this', this')\}$$
$$\texttt{ttail} := \texttt{this.tail}$$
$$sh_4 = \left\{ \begin{array}{c} (out, out), (this, this), (this, this'), (this', this'), \\ (ttail, this), (ttail, this'), (ttail, ttail) \end{array} \right\}$$
$$\texttt{if ttail} = \texttt{null then} \; \{\} \; \texttt{else out.tail} := \texttt{ttail.clone}()$$
$$sh_5 = sh_4.$$

We have $(out, this) \notin sh_5$ *i.e.,* our analysis guarantees now that *this* does not share with the result of `clone`. Note that $sh_5|_{\{out, this'\}} = \{(out, out), (this', this')\}$ *i.e.,* I is a fixpoint of the transformer of Definition 33.

6 Conclusions

We have equipped our new abstract domain Sh for pair-sharing analysis with abstract operations which allow us to show a simple example of analysis (Example 31). We know that some of these operations are not optimal, so there is space for improvement. Moreover, we still miss an implementation and, hence, an actual evaluation. We plan to implement Sh as an abstract domain for the Julia analyser [15], for which we already implemented 8 other abstract domains. We will use binary decision diagrams [3] to represent the denotational transfer functions over Sh of Figures 3 and 4. Exceptions are automatically transformed by Julia into branches in the program's control-flow, so they can be easily embedded in our sharing analysis as we already did for other static analyses.

References

[1] R. Bagnara, P. M. Hill, and E. Zaffanella. Set-Sharing is Redundant for Pair-Sharing. *Theoretical Computer Science*, 277(1–2):3–46, April 2002.

[2] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-Semantics Approach: Theory and Applications. *Journal of Logic Programming*, 19/20:149–197, 1994.

[3] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
[4] F. Bueno and M. J. García de la Banda. Set-Sharing Is Not Always Redundant for Pair-Sharing. In Y. Kameyama and P. J. Stuckey, editors, *Proc. of FLOPS'04*, volume 2998 of *Lecture Notes in Computer Science*, pages 117–131, Nara, Japan, April 2004. Springer-Verlag.
[5] J. D. Choi, M. Burke, and P. Carini. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. In *Proc. of the 20th Symposium on Principles of Programming Languages (POPL)*, pages 232–245, Charleston, South Carolina, January 1993. ACM.
[6] A. Cortesi and G. Filé. Abstract Interpretation of Logic Programs: An Abstract Domain for Groundness, Sharing, Freeness and Compoundness Analysis. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 52–61, Yale University, New Haven, Connecticut, USA, June 1991.
[7] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.
[8] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.
[9] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. In W. G. Olthoff, editor, *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, volume 952 of *LNCS*, pages 77–101, Århus, Denmark, August 1995. Springer.
[10] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent AND Parallelism. *Journal of Logic Programming*, 13(2 & 3):291–314, 1992.
[11] A. King. Pair-Sharing over Rational Trees. *Journal of Logic Programming*, 46(1–2):139–155, December 2000.
[12] V. Lagoon and P. J. Stuckey. Precise Pair-Sharing Analysis of Logic Programs. In *Proc. of the 4th international ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 99–108, Pittsburgh, PA, USA, October 2002. ACM.
[13] I. Pollet, B. Le Charlier, and A. Cortesi. Distinctness and Sharing Domains for Static Analysis of Java Programs. In *Proc. of the 25th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 77–98, Budapest, Hungary, June 2001.
[14] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. Available at `www.sci.univr.it/~spoto/papers.html`, 2005.
[15] F. Spoto. The JULIA Static Analyser. `www.sci.univr.it/~spoto/julia`, 2004.
[16] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proc. of the 23th ACM Symposium on Principles of Programming Languages (POPL)*, pages 32–41, St. Petersburg Beach, Florida, USA, January 1996.
[17] F. Tip and J. Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms. In *Proc. of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 35(10) of *SIGPLAN Notices*, pages 281–293, Minneapolis, Minnesota, USA, October 2000. ACM.
[18] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.

Exploiting Sparsity in Polyhedral Analysis

Axel Simon and Andy King

Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK
{a.simon, a.m.king}@kent.ac.uk

Abstract. The intrinsic cost of polyhedra has lead to research on more tractable sub-classes of linear inequalities. Rather than committing to the precision of such a sub-class, this paper presents a projection algorithm that works directly on any sparse system of inequalities and which sacrifices precision only when necessary. The algorithm is based on a novel combination of the Fourier-Motzkin algorithm (for exact projection) and Simplex (for approximate projection). By reformulating the convex hull operation in terms of projection, conversion to the frame representation is avoided altogether. Experimental results conducted on logic programs demonstrate that the resulting analysis is efficient and precise.

1 Introduction

Recently there has been much interest in so-called weakly relational domains [7,25,29] that trade the precision of operations on systems of linear inequalities for improved tractability. These domains seek to address the scalability problems of the polyhedral domain [8] whose operations are inherently exponential, irrespective of the algorithms used to implement them: Chandru *et al.* [6] showed that eliminating variables from a system of inequalities can increase the number of inequalities exponentially; Benoy *et al.* [2] showed that polytopes (bounded polyhedra) exist whose convex hull is exponential in the number of inequalities defining the input polytopes. Exponential growth also can arise when converting into the frame representation, which is the classical approach for computing the convex hull and projection [21]. Consider, for example, the convex hull of two n-dimensional hypercubes where one is translated along one axis. The frame consists of 2^n vertices for each hypercube. However, the resulting hull can be represented by $2n$ inequalities, just as the inputs. A natural question is whether there are faster methods to calculate the convex hull that do not convert the input polyhedra into their frame representation and that over-approximate the output polyhedron in case the resulting set of inequalities has exponential size.

One answer to this question is represented by the class of weakly relational domains where inequalities are restricted in order to prevent exponential growth. The Octagon domain [25] uses inequalities of the form $\pm x_i \pm x_j \leq c_{i,j}$ where x_i and x_j are variables and $c_{i,j}$ is a constant. In this domain the convex hull reduces to calculating the element-wise maximum of two matrices. The Octagon domain was generalised into the Octahedron domain [7], allowing more than two variables with zero or unary coefficients whilst maintaining a hull operation that

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 336–351, 2005.
© Springer-Verlag Berlin Heidelberg 2005

is polynomial in the number of variables. Finally, the two variables per inequality (TVPI) domain [29] allows arbitrary coefficients. This domain stores a planar polyhedron for each variable pair and employs a convex hull algorithm that operates on planar polyhedra [28]. All these domains employ a closure operation to propagate information between inequalities. Even incremental versions of these closure operations are quadratic, hence Blanchet *et al.* advocate a packing strategy when analysing large-scale programs [3]. They keep a set of Octagons, each describing relationships between variables occurring in a pack. Packs can overlap and are chosen by examining which variables occur in the same program statement. Packs are determined up front and hence packing variables is a commitment to a fixed degree of precision. Interestingly their program can be verified with packs that contain no more than four variables on average which suggests that useful inequalities contain relatively few variables. Halbwachs *et al.* also exploit the loose coupling of variables by partitioning the variable set into non-overlapping groups [13]. By applying the standard domain operations independently to each partition (rather than over the whole set of variables) useful speedups are obtained.

This paper shows how to exploit the fact that a given variable typically occurs in only a few inequalities. The key observation is that projection on these sparse systems can be realised efficiently by carefully applying the Fourier-Motzkin method [26]. We restrict the size of the output and the intermediate systems to be no larger than that of the input system which avoids exponential growth in the number of inequalities, thereby providing a performance guarantee. Surprisingly, even with this draconian size restriction the vast majority of variables can be eliminated. In the remaining cases we use Simplex to approximate the projection space by combining those inequalities that still contain uneliminated variables. Our method creates one inequality in the projection space for each call to Simplex. Simplex is called once for each remaining inequality which ensures that the final system is no larger than the original. This second stage over-approximates the projection (if applied at all). In terms of complexity, Fourier-Motzkin eliminates n variables in $O(nm)$ time where m is the number of inequalities. When variables remain to be eliminated, no more than m Simplex queries are performed where each query operates over m dimensions and n inequalities (note that n and m are exchanged). This method is attractive because, although Simplex is not a polynomial-time algorithm, the number of pivoting steps is about linear in the number of dimensions [27] and each pivoting step is in $O(nm)$ for the Simplex method in the tableau form. In fact, the average number of steps is polynomial [4]. To complete the set of domain operations, convex hull is recast in terms of projection [2] so that the frame representation is avoided altogether.

The remainder of the paper is structured as follows: After Section 2 introduces necessary mathematical notation, Section 3 presents techniques for using Fourier-Motzkin variable elimination for sparse inequality systems. Section 4 describes an approximation to projection. Section 5 describes how these efficient projection algorithms can be used to calculate convex hull. The paper finishes with sections on performance evaluation and related work before concluding.

2 Preliminaries

Let $Lin_X^=$ and $Lin_X^\leq$ denote the set of linear equalities and inequalities, respectively, defined over a finite set of variables X. Elements of $Lin_X^=$ and $Lin_X^\leq$ take the form of $\boldsymbol{c} \cdot \boldsymbol{y} = b$ and $\boldsymbol{c} \cdot \boldsymbol{y} \leq b$ where $|\boldsymbol{c}| = |\boldsymbol{y}|$, $b \in \mathbb{Z}$ and the elements of $\boldsymbol{c}$ and $\boldsymbol{y}$ are drawn from $\mathbb{Z}$ and X respectively. Furthermore let Con_X denote the set of all finite subsets of $Lin_X^= \cup Lin_X^\leq$ and $Ineq_X$ the set of all finite subsets of $Lin_X^\leq$. The set of real solutions for $\boldsymbol{c} \cdot \boldsymbol{y} \leq b$ is defined by:

$$soln_{\boldsymbol{x}}^{\mathbb{R}}(\boldsymbol{c} \cdot \boldsymbol{y} \leq b) = \left\{ \langle r_1, \ldots r_n \rangle \in \mathbb{R}^n \;\middle|\; \begin{array}{l} \boldsymbol{c} \cdot \langle r'_1, \ldots r'_m \rangle \leq b \quad \wedge \\ r_i = r'_j \text{ for all } x_i = y_j \end{array} \right\}$$

where $\boldsymbol{x} = \langle x_1, \ldots x_n \rangle$ and $\boldsymbol{y} = \langle y_1, \ldots y_m \rangle$. The real solution set for $\boldsymbol{c} \cdot \boldsymbol{y} = b$ is defined likewise and for any linear system $E \in Con_X$ the real solution set for E is defined $soln_{\boldsymbol{x}}^{\mathbb{R}}(E) = \cap_{e \in E} soln_{\boldsymbol{x}}^{\mathbb{R}}(e)$. Two linear systems $E_1, E_2 \in Con_X$ are partially ordered by the subset relation on their solution sets, that is, $E_1 \models_{\mathbb{R}} E_2$ iff $soln_{\boldsymbol{x}}^{\mathbb{R}}(E_1) \subseteq soln_{\boldsymbol{x}}^{\mathbb{R}}(E_2)$ where $var(\boldsymbol{x}) = var(E_1) \cup var(E_2)$ and $var(o)$ denotes the set of variables in a syntactic object o. The set of integer solutions $soln_{\boldsymbol{x}}^{\mathbb{Z}}$ can be defined analogously to $soln_{\boldsymbol{x}}^{\mathbb{R}}$ to induce a different partial order $E_1 \models^{\mathbb{Z}} E_2$. The ordering $\models^{\mathbb{R}}$ over-approximates $\models^{\mathbb{Z}}$ in the sense that if $E_1 \models^{\mathbb{R}} E_2$ then $E_1 \models^{\mathbb{Z}} E_2$; this is convenient in applications that are concerned with integral entities because (domain) operations associated with the ordering $\models^{\mathbb{R}}$ are more tractable than those induced by $\models^{\mathbb{Z}}$ [27]. Thus, henceforth, $\models$ and $soln_{\boldsymbol{x}}$ will abbreviate $\models^{\mathbb{R}}$ and $soln_{\boldsymbol{x}}^{\mathbb{R}}$ respectively. The predicate $sat \subseteq Con_X$ is defined so that $sat(E)$ holds iff $soln_{\boldsymbol{x}}(E) \neq \emptyset$ where $var(\boldsymbol{x}) = var(E)$. Finally, let *false* denote a particular system $E \in Con_X$ such that $sat(\mathit{false})$ does not hold.

3 Fourier-Motzkin Projection

Eliminating a variable from two equalities by scaling and adding them is a well known principle that is attributed to Gauss. Fourier refined this elimination strategy to pairs of inequalities. The basic observation is that inequalities may only be scaled by non-negative numbers which implies that the coefficients of the variable to be eliminated must have opposing signs in the two inequalities. His method was later elaborated on by Motzkin and henceforth it will be referred to as the Fourier-Motzkin variable elimination. While this algorithm has been thoroughly studied [11,15,18], little practical work has been reported on combining different refinements. This section presents strategies that are useful for program analysis – each strategy is reported in a separate sub-section.

Algorithm 1. presents the basic Fourier-Motzkin algorithm to remove a variable $x_r \in X$ from a system of inequalities $E \in Ineq_X$. E is partitioned into E^+, E^r and E^-, corresponding to inequalities that have positive, zero and negative coefficient for x_r. E^r is augmented to obtain the projection by combining positive multiples of pairs of inequalities drawn from E^+ and E^-. The variable x_r is eliminated since the coefficient of x_r in each inequality added to

Algorithm 1. Fourier-Motzkin $\mathit{fourier}(x_r, E)$

Require: $x_r \in X$, $E \in \mathit{Ineq}_X$
  $\langle E^+, E^r, E^- \rangle \leftarrow \langle \emptyset, \emptyset, \emptyset \rangle$
  for $\boldsymbol{a} \cdot \boldsymbol{x} \leq c \in E$ **do**
    if $\pi_r(\boldsymbol{a}) = 0$ **then**
      $E^r \leftarrow E^r \cup \{\boldsymbol{a} \cdot \boldsymbol{x} \leq c\}$
    else if $\pi_r(\boldsymbol{a}) > 0$ **then**
      $E^+ \leftarrow E^+ \cup \{\boldsymbol{a} \cdot \boldsymbol{x} \leq c\}$
    else
      $E^- \leftarrow E^- \cup \{\boldsymbol{a} \cdot \boldsymbol{x} \leq c\}$
  for $\boldsymbol{a}^+ \cdot \boldsymbol{x} \leq c^+ \in E^+$ **do**
    for $\boldsymbol{a}^- \cdot \boldsymbol{x} \leq c^- \in E^-$ **do**
      $\boldsymbol{a} \cdot \boldsymbol{x} \leq c \leftarrow \mathit{simplify}(\ (\pi_r(\boldsymbol{a}^+)\boldsymbol{a}^- + |\pi_r(\boldsymbol{a}^-)|\boldsymbol{a}^+) \cdot \boldsymbol{x} \leq (\pi_r(\boldsymbol{a}^+)c^- + |\pi_r(\boldsymbol{a}^-)|c^+)\ )$
      if $\boldsymbol{a} \neq \boldsymbol{0}$ **then**
        $E^r \leftarrow E^r \cup \{\boldsymbol{a} \cdot \boldsymbol{x} \leq c\}$
      else if $c < 0$ **then**
        return *false*
  return E^r

E^r is $\pi_r(\boldsymbol{a}^+)\pi_r(\boldsymbol{a}^-) + |\pi_r(\boldsymbol{a}^-)|\pi_r(\boldsymbol{a}^+) = 0$ where $\boldsymbol{a}^+ \in E^+$ and $\boldsymbol{a}^- \in E^-$ and $\pi_i(\langle a_1, \ldots a_n \rangle) = a_i$. Note that a generated inequality might take the form $\boldsymbol{0} \cdot \boldsymbol{x} \leq c$. If $c < 0$, the original system E is unsatisfiable and *false* is returned as the projection, otherwise the inequality is a tautology [20] and is discarded.

3.1 Simplification

To identify equivalent inequalities, a unique representation is desirable. The *simplify* function presented as Algorithm 2. is designed to remove common factors from a newly generated inequality. The algorithm is generic in the sense that it supports equalities, so that it is also applicable in Gaussian elimination (as discussed in Section 3.5). In both cases the function is the identity if all coefficients are zero since this represents either a tautology or a contradiction. Otherwise an inequality is divided by the greatest common denominator of its coefficients. Note that dividing the constant might not result in an integral number and therefore the result is rounded down. This is sound only if integer entities are represented. In the case of equalities, the assignment $g \leftarrow c$ ensures that the *gcd* calculation also considers the constant so that the division has no remainder, thereby guaranteeing that the exact equality relationship is preserved.

3.2 Variable Selection

Whenever a set of variables $Y = \{y_1, \ldots y_n\}$ needs to be projected out, the Fourier-Motzkin algorithm can be applied iteratively by setting $E_0 = E$ and $E_i = \mathit{fourier}(y_i, E_{i-1})$. In each step, $|E_i^+| + |E_i^-|$ inequalities are removed from E_i and $|E_i^+||E_i^-|$ are added. Hence the growth in each step is in $O(|E_i|^2)$ and the number of inequalities in the final system E_n is in $O(|E|^{2^n})$ which prohibits

Algorithm 2. Simplification $simplify(\boldsymbol{a} \cdot \boldsymbol{x} \odot c)$ where $\odot \in \{\leq, =\}$

```
if a = 0 then
  return a · x ⊙ c
if ⊙ ∈ {≤} then
  g ← 0
else
  g ← c
for a_i ∈ a do
  if a_i ≠ 0 then
    if g = 0 then
      g ← a_i
    else
      g ← gcd(g, a_i)
return (a/g) · x ⊙ ⌊c/g⌋
```

Algorithm 3. Select variable $select(Y, E)$

```
Require: E ∈ Ineq_X, Y ⊆ X
  ⟨p_1, ... p_|X|⟩ ← 0
  ⟨m_1, ... m_|X|⟩ ← 0
  for a · x ≤ c ∈ E do
    for i ∈ {1, ... |X|} do
      if π_i(a) > 0 then
        p_i ← p_i + 1
      else if π_i(a) < 0 then
        m_i ← m_i + 1
  bestGrowth ← |E|^2
  for x_i ∈ Y do
    growth ← p_i m_i − (p_i + m_i)
    if growth < bestGrowth then
      bestGrowth ← growth
      bestVar ← x_i
  return ⟨bestGrowth, bestVar⟩
```

direct use of this method even for projecting out a few variables. A standard rule [11] suggests delaying the growth of the intermediate systems by always eliminating the variable that minimises $|E_i^+||E_i^-| - (|E_i^+| + |E_i^-|)$. Algorithm 3. calculates how many positive and negative coefficients each variable has in the given inequality system E. It returns the variable x_r such that applying Fourier-Motzkin elimination will result in minimal growth.

3.3 Complete Redundancy Removal

Each Fourier-Motzkin step may introduce redundant inequalities. Algorithm 4. uses the Simplex method to check every inequality for redundancy. The function $simplex(\boldsymbol{a}, \boldsymbol{x}, E)$ calculates a vector $\boldsymbol{m}$ that maximises $\boldsymbol{a} \cdot \boldsymbol{x}$ subject to the linear inequalities in E. Running *compress* after each Fourier-Motzkin elimination step

Algorithm 4. Complete Redundancy Removal *compress*(E)

Require: $E \in Ineq_X$
  if $\neg sat(E)$ **then**
    return *false*
  for $\boldsymbol{a} \cdot \boldsymbol{x} \leq c \in E$ **do**
    $\boldsymbol{m} \leftarrow simplex(\boldsymbol{a}, \boldsymbol{x}, E \setminus \{\boldsymbol{a} \cdot \boldsymbol{x} \leq c\})$
    if $\boldsymbol{m} \cdot \boldsymbol{a} \leq c$ **then**
      $E \leftarrow E \setminus \{\boldsymbol{a} \cdot \boldsymbol{x} \leq c\}$
  return E

Algorithm 5. Quasi-Syntactic Redundancy Removal *quasi*(E)

Require: $E \in Ineq_X$
  while $\{\boldsymbol{a_1} \cdot \boldsymbol{x} \leq c_1, \boldsymbol{a_2} \cdot \boldsymbol{x} \leq c_2\} \subseteq E \wedge \boldsymbol{a_1} = \boldsymbol{a_2}$ **do**
    if $c_1 > c_2$ **then**
      $E \leftarrow E \setminus \{\boldsymbol{a_1} \cdot \boldsymbol{x} \leq c_1\}$
    else
      $E \leftarrow E \setminus \{\boldsymbol{a_2} \cdot \boldsymbol{x} \leq c_2\}$
  return E

is prohibitively expensive and therefore it is desirable to only apply *compress* when more lightweight redundancy removal algorithms fail to constrain growth.

3.4 Quasi-Syntactic Redundancy Removal

Lassez *et al.* identify several classes of redundant inequalities that can be detected by purely syntactic means [20]. For instance, inequalities with identical coefficients are called syntactically redundant (if the constant is equal) and quasi-syntactically redundant (if the constants differ). Given a pair of quasi-syntactic redundant inequalities, only the one with the smaller constant needs to be retained. Algorithm 5. removes both classes of redundancy by examining pairs of inequalities. In practise, inequalities can be sorted lexicographically by their coefficients which allows the algorithm to run in $O(|E| \log |E|)$.

3.5 Equality Removal

Rather than modelling an equality as two opposing inequalities, it is more prudent to retain equalities that arise during the analysis and precede the Fourier-Motzkin elimination with a Gaussian elimination phase. Algorithm 6. takes as input the system of equalities and inequalities E and the set of variables Y that are to be eliminated. It returns as output a triple consisting of a set of variables that remain to be eliminated, a set of equalities P in the projection space, and a system of inequalities that still retain variables to be eliminated. The algorithm iterates as long as there remains an equality $\boldsymbol{a} \cdot \boldsymbol{x} = c \in E$. If there exists a coefficient $\pi_i(\boldsymbol{a}) \neq 0$ and $\pi_i(\boldsymbol{x}) \in Y$ then Gaussian elimination is performed on all inequalities and remaining equalities that contain a non-zero coefficient for

Algorithm 6. Equality Removal $gauss(Y, E)$

```
Require: E ∈ Con_X, Y ⊆ X
  P ← ∅
  while a · x = c ∈ E do
    E ← E \ {a · x = c}
    s ← −1
    for x_i ∈ Y do
      if π_i(a) ≠ 0 then
        s ← i
    if s = −1 then
      P ← P ∪ {a · x = c}
      s ← i such that π_i(a) ≠ 0
    Y ← Y \ {x_s}
    if π_s(a) < 0 then
      ⟨a, c⟩ ← ⟨−a, −c⟩
    E' ← ∅
    for b · x ⊙ d ∈ E where ⊙ ∈ {≤, =} do
      if π_s(b) = 0 then
        E' ← E' ∪ {b · x ⊙ d}
      else
        e · x ⊙ f ← simplify( (a_s b − b_s a) · x ⊙ (a_s d − b_s c) )
        if e = 0 then
          if (⊙ ∈ {≤} ∧ f < 0) ∨ (⊙ ∈ {=} ∧ f ≠ 0) then
            return ⟨Y, false, P⟩
        else
          E' ← E' ∪ {e · x ⊙ f}
    E ← E'
  return ⟨Y, E, P⟩
```

$\pi_i(\boldsymbol{x})$. Since $\pi_i(\boldsymbol{x})$ is to be eliminated, the equality is then discarded. Alternatively, if there is no variable $\pi_i(\boldsymbol{x}) \in Y$ with $\pi_i(\boldsymbol{a}) \neq 0$ then the equality is part of the projection space P. Observe that each iteration of the while loop makes progress in the sense that it reduces the set of variables that appear in E.

The value of applying Gaussian elimination is fourfold: (1) it avoids reformulating each equality as two inequalities; (2) it reduces the number of inequalities that Fourier-Motzkin is applied to; (3) it reduces the number of variables that remain to be eliminated and perhaps most subtly (4) it increases the number of inequalities that can be identified as quasi-syntactically redundant. The last point stems from the observation that substituting an equality into a system often reformulates one inequality to the extent that it becomes quasi-syntactically redundant with respect to another [20]. This motivates the substitution of all equalities, even those that do not contain variables to be eliminated.

3.6 Combining All Strategies

This section composes the strategies previously presented so as to ensure tractability even in those pathological cases when the size of the projection is expo-

nential [2]. The overall projection method is presented as Algorithm 7. and takes a linear system E and a set of variables Y that are to be eliminated. The algorithm applies Gaussian elimination to produce a system of inequalities E no larger than the initial input. Fourier-Motzkin elimination is then performed, which is interleaved with quasi-syntactic redundancy removal, until no more variables can be eliminated without exceeding the preset limit. Due to sparsity, a variable will often only appear once with a certain polarity (say with a positive coefficient). In this case the number of inequalities removed will be $|E^+| + |E^-| = 1 + n$ and the number of newly created inequalities is at most $|E^+||E^-| = n$ which makes the system shrink. Another frequently occurring case is that of $|E^+| = |E^-| = 2$. If the limit is exceeded, complete redundancy removal is activated in an attempt to remove enough inequalities to resume Fourier-Motzkin. At this stage, the limit is further reduced to $|E|$. This is good practise since E is usually reduced considerably. Finally, if Fourier-Motzkin cannot be reapplied and variables remain to be eliminated, the system E is partitioned into those inequalities E' that contain variables in Y and into those in P that do not. The projection of the set E' is approximated by the extreme point projection which is presented next.

4 Extreme Point Projection

While the Fourier-Motzkin method works well on sparse systems, Huynh *et al.* [14] propose using the extreme point method of Lassez [19] for dense systems. This method can find inequalities in the projection space incrementally, thereby enabling the projection to be approximated with a limited number of inequalities. To illustrate the method, consider eliminating the variables Y from a linear system $E = \{\boldsymbol{a}_1 \cdot \boldsymbol{x} \leq c_1, \ldots \boldsymbol{a}_n \cdot \boldsymbol{x} \leq c_n\}$. W.l.o.g., let $Y = var(\boldsymbol{y})$ and

$$\begin{pmatrix} \boldsymbol{a}_1 \\ \hline \vdots \\ \hline \boldsymbol{a}_n \end{pmatrix} = (A|B) \qquad \begin{pmatrix} x_1 \\ \hline \vdots \\ \hline x_m \end{pmatrix} = \begin{pmatrix} \boldsymbol{y} \\ \hline \boldsymbol{z} \end{pmatrix} \qquad \boldsymbol{c} = \begin{pmatrix} c_1 \\ \hline \vdots \\ \hline c_n \end{pmatrix}$$

where $\boldsymbol{y}$ and $\boldsymbol{z}$ are column vectors and $\boldsymbol{x} = \langle x_1, \ldots x_m \rangle$. Then $A\boldsymbol{y} + B\boldsymbol{z} \leq \boldsymbol{c}$ is equivalent to E and the problem of calculating an inequality in the projection space reduces to finding non-negative linear combinations $\boldsymbol{\lambda} \in \mathbb{R}^n$ of rows of A such that $\boldsymbol{\lambda} A = \boldsymbol{0}$. Then $\boldsymbol{\lambda}(A\boldsymbol{y} + B\boldsymbol{z}) \leq \boldsymbol{\lambda}\boldsymbol{c}$ and $\boldsymbol{\lambda}(A\boldsymbol{y} + B\boldsymbol{z}) = \boldsymbol{\lambda} B\boldsymbol{z}$ hence $(\boldsymbol{\lambda} B)\boldsymbol{z} \leq \boldsymbol{\lambda}\boldsymbol{c}$ which yields an inequality in the projection space. The vector $\boldsymbol{\lambda} = \boldsymbol{0}$ is the trivial solution to the system $\boldsymbol{\lambda} A = \boldsymbol{0}$ yielding a tautology. Observe that if $\boldsymbol{\lambda} \in \mathbb{R}^n$ is a solution to $\boldsymbol{\lambda} A = \boldsymbol{0}$ and $\{\lambda_i \geq 0 \mid \lambda_i \in \boldsymbol{\lambda}\}$, then so is $s\boldsymbol{\lambda}$ where $s \in \mathbb{R}$ is any non-negative scalar. Hence, w.l.o.g., we can enforce the constraint $\lambda_1 + \ldots + \lambda_n = 1$. The set of all extreme points of the bounded space, given by $\boldsymbol{\lambda} A = \boldsymbol{0}$, $\{\lambda_i \geq 0 \mid \lambda_i \in \boldsymbol{\lambda}\}$ and $\lambda_1 + \ldots + \lambda_n = 1$, corresponds to the exact projection which potentially contains an exponential number of inequalities. To give a performance guarantee, we only enumerate $|E|$ extreme points thereby ensuring that the number of inequalities does not grow beyond the set limit.

Since extreme point enumeration does not consider the B matrix, two extreme points $\boldsymbol{\lambda}_a \neq \boldsymbol{\lambda}_b$ might produce the same coefficient vector $\boldsymbol{\lambda}_a B = \boldsymbol{\lambda}_b B$

Algorithm 7. Projection $project(Y, E)$

```
Require: E ∈ Con_X, Y ⊆ X
  ⟨Y, E, P⟩ ← gauss(Y, E)
  if ¬sat(E) then
    return false
  limit ← |E|
  ⟨g, x_i⟩ ← select(Y, E)
  while Y ≠ ∅ ∧ |E| + g ≤ limit do
    E ← fourier(x_i, E)
    if E = false then
      return false
    E ← quasi(E)
    Y ← Y \ {x_i}
    ⟨g, x_i⟩ ← select(Y, E)
    if |E| + g > limit then
      E ← compress(E)
      if E = false then
        return false
      limit ← |E|
      ⟨g, x_i⟩ ← select(Y, E)
  if Y = ∅ then
    return compress(E ∪ P)
  E′ ← ∅
  for a · x ≤ c ∈ E do
    if ∃x_i ∈ Y.π_i(a) ≠ 0 then
      E′ ← E′ ∪ {a · x ≤ c}
    else
      P ← P ∪ {a · x ≤ c}
  return compress(extreme(Y, E′) ∪ P)
```

such that one of the resulting inequalities will be quasi-syntactically redundant. Kohler [18] observed that if the set of indices containing zero coefficients in $\boldsymbol{\lambda}_a$ is a strict superset of those of $\boldsymbol{\lambda}_b$, then the latter leads to a redundant inequality. This observation can be exploited by maximising the number of zero coefficients in each $\boldsymbol{\lambda}$ which is the indirect result of running a linear program that maximises a specific $\lambda_i \in \boldsymbol{\lambda}$. Algorithm 8. formalises this heuristic. As a final comment, note that Fourier-Motzkin elimination can be seen as a special case of the extreme point method where A only contains one column (and hence one variable to eliminate). The extreme points are those solutions that combine exactly one positive row with one negative row in A.

5 Convex Hull via Projection

The convex hull operation takes as input two inequality sets $E_1, E_2 \in Con_X$ and produces as output an $E \in Con_X$ such that $soln_{\boldsymbol{x}}(E_i) \subseteq soln_{\boldsymbol{x}}(E)$, $soln_{\boldsymbol{x}}(E)$ is minimal and $var(\boldsymbol{x}) = var(E_1 \cup E_2)$. For purpose of exposition, let E_1 and E_2

Algorithm 8. Extreme-Point Projection $\mathit{extreme}(Y, E)$

Require: $E \in \mathit{Ineq}_X$, $Y \subseteq X$

$(A|B) \leftarrow \begin{pmatrix} \boldsymbol{a}_1 \\ \hline \vdots \\ \hline \boldsymbol{a}_n \end{pmatrix}$ **where** $(\boldsymbol{a}_i \cdot \boldsymbol{x} \leq c_i) \in E, \mathit{var}(\boldsymbol{y}) = Y$ **and** $A\boldsymbol{y} + B\boldsymbol{z} \leq \begin{pmatrix} c_1 \\ \hline \vdots \\ \hline c_n \end{pmatrix}$

$\Lambda \leftarrow \boldsymbol{\lambda}A = 0 \cup \{\lambda_i \geq 0 \mid \lambda_i \in \boldsymbol{\lambda}\} \cup \{\sum \lambda_i = 1\}$

$E' \leftarrow \emptyset$

for $\boldsymbol{f} \in \langle 1, 0, \ldots 0\rangle, \langle 0, 1, \ldots 0\rangle \ldots \langle 0, 0, \ldots 1\rangle$ **do**

  $\boldsymbol{m} \leftarrow \mathit{simplex}(\boldsymbol{f}, \boldsymbol{\lambda}, \Lambda)$

  $e \leftarrow \mathit{simplify}(\boldsymbol{m}B \leq \boldsymbol{m} \cdot \langle c_1, \ldots c_n\rangle)$

  $E' \leftarrow E' \cup \{e\}$

return E'

be represented in matrix form as $A_i\boldsymbol{x} \leq \boldsymbol{c}_i$, $i = 1, 2$ (with the equalities in E_i expressed as two rows in A_i). The smallest convex set of points P that includes $\mathit{soln}_{\boldsymbol{x}}(E_1) \cup \mathit{soln}_{\boldsymbol{x}}(E_2)$ is given by

$$P = \left\{ \boldsymbol{x} \,\middle|\, \begin{array}{lll} \boldsymbol{x} = \sigma_1\boldsymbol{x}_1 + \sigma_2\boldsymbol{x}_2 & \wedge\, \sigma_1 + \sigma_2 = 1 & \wedge\, \sigma_1 \geq 0\, \wedge \\ A_1\boldsymbol{x}_1 \leq \boldsymbol{c}_1 & \wedge\, A_2\boldsymbol{x}_2 \leq \boldsymbol{c}_2 & \wedge\, \sigma_2 \geq 0 \end{array} \right\}.$$

To avoid the non-linearity $\boldsymbol{x} = \sigma_1\boldsymbol{x}_1 + \sigma_2\boldsymbol{x}_2$, the system can be relaxed by setting $\boldsymbol{y}_1 = \sigma_1\boldsymbol{x}_1$ and $\boldsymbol{y}_2 = \sigma_2\boldsymbol{x}_2$ so that $\boldsymbol{x} = \boldsymbol{y}_1 + \boldsymbol{y}_2$ and $A_i\boldsymbol{y}_i \leq \sigma_i\boldsymbol{c}_i$ to define:

$$P' = \left\{ \boldsymbol{x} \,\middle|\, \begin{array}{lll} \boldsymbol{x} = \boldsymbol{y}_1 + \boldsymbol{y}_2 & \wedge\, \sigma_1 + \sigma_2 = 1 & \wedge\, \sigma_1 \geq 0\, \wedge \\ A_1\boldsymbol{y}_1 \leq \sigma_1\boldsymbol{c}_1 & \wedge\, A_2\boldsymbol{y}_2 \leq \sigma_2\boldsymbol{c}_2 & \wedge\, \sigma_2 \geq 0 \end{array} \right\}.$$

Note that, although $P \subseteq P'$, in general $P \neq P'$ since P might not be topologically closed whereas P' is represented by a set of (non-strict) inequalities and therefore is closed. In fact projecting out σ_i and the variables in $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ yields a system E representing the closure of the convex hull of the two input polyhedra E_1 and E_2 as is formally proved in [2]. Henceforth let $\mathit{hull}(E_1, E_2) = E$ encapsulate this computational tactic for calculating the convex hull. Since entailment can be realised straightforwardly with Simplex, this section completes the suite of polyhedral domain operations without recourse to the frame representation.

6 Performance Evaluation

In order to assess the precision and efficiency of the domain operations reported thus far, the algorithms have been integrated into an argument size analyser [10]. The analysis is key to termination checking [10], termination inference [24], control generation [17] and determinacy inference [23]. The last application uses argument size relationships that are synthesised for each clause in the program to infer a determinacy condition for each predicate that, if satisfied by a call, guarantees that there is at most one computed answer for that call and that the answer is produced only once if ever. The value of this analysis quickly degrades unless three variable inequalities can be inferred (see [23, Section 2.2]) which precludes the use of octagons [25] or the TVPI domain [29].

6.1 Argument-Size Analysis of Logic Programs

This section summarises the essential details of an argument-size analysis. The analysis abstracts the standard T_P [22] operator of a logic program P. In this presentation, T_P is defined for clauses of the form $p(\boldsymbol{x}) \leftarrow H, p_1(\boldsymbol{x}_1), \ldots p_n(\boldsymbol{x}_n)$ where $\boldsymbol{x}$ and $\boldsymbol{x}_i$ are vectors of variables, H is a finite (possibly empty) set of Herbrand equations $\{s_1 = t_1, \ldots s_n = t_n\}$ and s_i and t_i are arbitrary terms. The set of unifiers of H is denoted by $\mathit{unify}(H)$. For a given clause c, the operator $T_c(I)$ maps one set of ground atoms I to another in the following manner:

$$T_c(I) = I \cup \left\{ \theta(p(\boldsymbol{x})) \;\middle|\; \begin{array}{c} c = p(\boldsymbol{x}) \leftarrow H, p_1(\boldsymbol{x}_1), \ldots p_n(\boldsymbol{x}_n) \;\wedge \\ \mathit{var}(\theta(c)) = \emptyset \qquad \wedge \\ \theta \in \mathit{unify}(H) \;\;\wedge\;\; \theta(p_i(\boldsymbol{x}_i)) \in I \end{array} \right\}$$

The condition $\mathit{var}(\theta(c)) = \emptyset$ ensures that the substitution θ grounds c, hence the atom $\theta(p(\boldsymbol{x}))$ is variable-free. The operator lifts to a program $P = \{c_1, \ldots c_n\}$ by defining $T_P(I) = I_n$ where $I_0 = I$ and $I_i = T_{c_i}(I_{i-1})$. Since T_P is monotonic and the computation domain of sets of ground atoms constitutes a complete lattice under the subset ordering, then $\mathrm{lfp}(T_P)$ exists which provides a convenient fixpoint formulation of the semantics of P [22].

Argument-size analysis aspires to find size invariants for each p that describe a tuple of terms $\boldsymbol{t}$ whenever $p(\boldsymbol{t}) \in \mathrm{lfp}(T_P)$. Size is quantified in terms of a norm that maps a ground term to a non-negative size. In our experiments we use the term-size norm $|.|_{\mathit{term\text{-}size}}$ [9] which is defined as follows:

$$|t|_{\mathit{term\text{-}size}} = \begin{cases} 1 + \sum_{i=1}^{n} |t_i|_{\mathit{term\text{-}size}} & \text{if } t = f(t_1, \ldots t_n) \wedge n > 0 \\ 0 & \text{otherwise} \end{cases}$$

The established approach to finding such invariants involves describing Herbrand (syntactic) equations with linear equations. Formally, a linear equation $\boldsymbol{c} \cdot \boldsymbol{x} = b$ describes $s = t$ with respect to $|.|$, denoted by $(\boldsymbol{c} \cdot \boldsymbol{x} = b) \propto_{|.|} (s = t)$, iff $|\theta(\boldsymbol{x})| \in \mathit{soln}_{\boldsymbol{x}}(\boldsymbol{c} \cdot \boldsymbol{x} = b)$ whenever θ is a grounding substitution for $s = t$ such that $\theta \in \mathit{unify}(\{s = t\})$ where $|\langle t_1, \ldots t_n \rangle| = \langle |t_1|, \ldots |t_n| \rangle$. Since $\propto_{|.|}$ is a relation, a natural question is whether there is a best description of a given Herbrand equation $s = t$. In fact, this is given by $e = \alpha_{|.|}(s = t)$ where $\alpha_{|.|}$ is defined such that e is the best abstraction with $e \propto_{|.|} (s = t)$. For the term-size norm, and more generally the class of semi-linear norms [5], the function $\alpha_{|.|}$ is well-defined. The mapping $\alpha_{|.|}$ extends to sets of Herbrand equations by $\alpha_{|.|}(H) = \{\alpha_{|.|}(s_i = t_i) \mid (s_i = t_i) \in H\}$.

Example 1. To illustrate, consider the equation $\mathtt{C} = \mathtt{succ}(\mathtt{N}) * \mathtt{pow}(\mathtt{X}, \mathtt{N})$ where $*$ is an infix functor. The linear equation $\mathtt{C} = 3 + \mathtt{X} + 2 * \mathtt{N}$ describes the Herbrand equation with respect to $|.|_{\mathit{term\text{-}size}}$. To see this, let θ be a grounding unifier of the Herbrand equation. Then $|\theta(\mathtt{C})|_{\mathit{term\text{-}size}} = |\mathtt{succ}(\theta(\mathtt{N})) * \mathtt{pow}(\theta(\mathtt{X}), \theta(\mathtt{N}))|_{\mathit{term\text{-}size}}$, hence: $|\theta(\mathtt{C})|_{\mathit{term\text{-}size}} = 1 + (1 + |\theta(\mathtt{N})|_{\mathit{term\text{-}size}}) + (1 + |\theta(\mathtt{X})|_{\mathit{term\text{-}size}} + |\theta(\mathtt{N})|_{\mathit{term\text{-}size}})$ Observe that the linear equation expresses the relative sizes of any ground instance of the variables C, X and N that satisfies the syntactic equation.

To capture linear invariants between the arguments of predicates, it is necessary to lift the $\models$ ordering on linear systems to atoms paired with linear systems as follows: $\langle p(\boldsymbol{x}_1), E_1\rangle \models \langle p(\boldsymbol{x}_2), E_2\rangle$ iff $soln_{\boldsymbol{x}_1}(E_1) \subseteq soln_{\boldsymbol{x}_2}(E_2)$. Observe that two pairs $\langle p(\boldsymbol{x}_1), E_1\rangle$ and $\langle p(\boldsymbol{x}_2), E_2\rangle$ that differ syntactically may express the same invariants, that is, $\langle p(\boldsymbol{x}_1), E_1\rangle \models \langle p(\boldsymbol{x}_2), E_2\rangle \models \langle p(\boldsymbol{x}_1), E_1\rangle$ yet $E_1 \neq E_2$. To express invariants between argument positions it is thus necessary to construct sets of syntactically different but equivalence pairs. (This is more than an aesthetic predilection since this construction simplifies the way formal arguments are matched against actual arguments.) Formally, equivalence is defined by $\langle p(\boldsymbol{x}_1), E_1\rangle \equiv \langle p(\boldsymbol{x}_2), E_2\rangle$ iff $\langle p(\boldsymbol{x}_1), E_1\rangle \models \langle p(\boldsymbol{x}_2), E_2\rangle \models \langle p(\boldsymbol{x}_1), E_1\rangle$ which, in turn, induces a notion of equivalence class. To simultaneously record the invariants that hold on different predicates, the ordering is further extended to sets of equivalence classes to obtain a preorder. Specifically, given two sets of equivalence classes I_1 and I_2, the preorder $\models$ is defined $I_1 \models I_2$ iff for all $[\langle p(\boldsymbol{x}), E_1\rangle]_\equiv \in I_1$ there exists $[\langle p(\boldsymbol{x}), E_2\rangle]_\equiv \in I_2$ such that $\langle p(\boldsymbol{x}), E_1\rangle \models \langle p(\boldsymbol{x}), E_2\rangle$. Sets of equivalence classes provide a computation domain for the following operator that simulates T_P in such as fashion so as to discover argument-size relationships. The operator is denoted by T_c^{CLP} since it operates in the domain of linear constraints. Like before, it is defined in a clause-wise fashion:

$$T_c^{\mathrm{CLP}}(I) = I \cup \left\{ [\langle p(\boldsymbol{x}), hull(F, F')\rangle]_\equiv \;\middle|\; \begin{array}{ll} c = p(\boldsymbol{x}) \leftarrow H, p_1(\boldsymbol{x}_1), \ldots p_n(\boldsymbol{x}_n) & \wedge \\ [\langle p_i(\boldsymbol{x}_i), E_i\rangle]_\equiv \in I \wedge [\langle p(\boldsymbol{x}), F\rangle]_\equiv \in I & \wedge \\ E = \alpha_{|.|}(H) \cup (\cup_{i=1}^n E_i) & \wedge \\ F' = project(var(c) \setminus var(\boldsymbol{x}), E) & \end{array} \right\}$$

This operator can be lifted to the level of a program $P = \{c_1, \ldots c_n\}$ by defining $T_P^{\mathrm{CLP}}(I) = I_n$ where $I_0 = I$ and $I_i = T_{c_i}^{\mathrm{CLP}}(I_{i-1})$. The computational domain is neither a complete lattice nor admits finite ascending chains. However, by adding a widening operator [8] a post-fixpoint can be finitely computed, that is, a set of equivalence classes I such that $T_P^{\mathrm{CLP}}(I) \models I$. Such a post-fixpoint faithfully describes the lfp of the original program in the following sense: if $T_P^{\mathrm{CLP}}(I) \models I$ and $p(\boldsymbol{t}) \in \mathrm{lfp}(T_P)$ then there exists $[\langle p(\boldsymbol{x}), E\rangle]_\equiv \in I$ such that $|\boldsymbol{t}| \in soln_{\boldsymbol{x}}(E)$. The proof is not given since it can be constructed straightforwardly by adapting proofs that have been reported elsewhere [12]. $T_{c_i}^{\mathrm{CLP}}$ provides a way to calculate a post-fixpoint in a bottom-up fashion by iterating and stabilising each strongly connected component (SCC) of the static call graph in turn. SCCs that contain a single, non-recursive clause can be evaluated exactly without a stability check.

6.2 Experimental Results

For simplicity, an argument-size analyser was implemented in SICStus Prolog 3.8.5 which comes equipped with a built-in Simplex solver. The analyser was applied to a range of standard Prolog benchmarks varying in size between 100 and 10000+ LOC. Figure 1 presents the analysis times in seconds when the analyser is run on a 2.40GHz PC with 512 MB of RAM running Windows XP with all modules compiled to so-called compactcode (interpreted bytecode). The

benchmark	LOC	vars approx'ed ratio	%	proj approx'ed ratio	%	sparsity size	system	vars	time
gabriel	114	0/186	0.0	0/60	0.0	5.6	10.4	1.4	0.06
browse	137	0/294	0.0	0/79	0.0	6.5	11.9	1.4	0.06
ime_v2-2-1	181	21/888	2.3	8/132	6.0	11.9	21.3	1.6	0.70
kalah	284	0/533	0.0	0/133	0.0	7.3	12.4	1.4	0.14
mastermind	311	0/352	0.0	0/89	0.0	6.3	12.2	1.4	0.11
sdda	331	4/432	0.9	2/137	1.4	6.2	11.0	1.4	0.11
press	349	14/802	1.7	7/215	3.2	6.5	11.9	1.5	0.31
trs	368	7/1651	0.4	5/209	2.3	12.2	21.4	1.6	0.37
peep	371	11/665	1.6	6/163	3.6	7.7	12.5	1.6	0.34
qplan	424	0/380	0.0	0/104	0.0	7.9	14.7	1.4	0.09
ga	437	0/479	0.0	0/87	0.0	10.7	20.0	1.4	0.17
read	442	4/844	0.4	2/213	0.9	7.7	15.4	1.3	0.23
simple_analyzer	488	5/1183	0.4	3/287	1.0	8.6	15.0	1.4	0.44
ann	503	9/1089	0.8	3/268	1.1	7.7	12.9	1.5	0.39
nbody	562	0/684	0.0	0/147	0.0	9.2	15.9	1.3	0.13
ili	582	6/1789	0.3	3/504	0.5	7.8	13.6	1.3	0.64
asm	594	1/761	0.1	1/217	0.4	7.2	12.3	1.3	0.24
nand	603	29/1356	2.1	6/240	2.5	11.1	19.8	1.4	1.53
bryant	670	22/1381	1.5	4/252	1.5	14.1	26.3	1.3	1.38
sim_v5-2	986	14/2923	0.4	8/840	0.9	6.0	11.2	1.4	0.88
peval	993	36/2709	1.3	18/719	2.5	9.7	17.3	1.3	1.79
sim	1071	0/2412	0.0	0/394	0.0	12.0	20.1	1.3	0.61
rubik	1229	0/1062	0.0	0/276	0.0	5.7	9.4	1.5	0.20
chat	4698	105/7917	1.3	50/1581	3.1	9.7	19.1	1.5	4.58
pl2wam	4775	96/4078	2.3	34/1020	3.3	8.0	13.4	1.5	3.20
lptp	7419	213/12525	1.7	81/3624	2.2	8.2	15.2	1.4	9.97
aqua_c	15026	493/32340	1.5	188/6292	2.9	10.3	19.5	1.5	27.59

Fig. 1. Timing and precision results

leftmost column records the time to actually calculate the size invariants and write the results to an output file (little variance was observed between different runs of the analyser). This excludes the time to read, parse and normalise the input program and compute the SCCs (which is a small and varying fraction of the analysis time). These experiments were conducted using the classic widening [8] but delaying its application within an SCC until 2 complete iterations had been computed. Performance figures for an argument size analysis have been reported for the cTI termination inference tool [24]. cTI realises its argument size analysis with the Parma Polyhedra Library (PPL version 0.5 [1]) and timings of 0.26s, 0.17s, 3.89s, 3.99s and 2.12s are reported for read, ann, chat, lptp and pl2wam – the largest five benchmarks that we have in common and which were publicly available. These experiments were also performed on a 2.4GHz PC with 512 MB of RAM, albeit running Linux, with widening activated after one SCC iteration. Repeating our experiments with this widening tactic gives times of 0.11s, 0.27s, 3.98s, 6.12s, 1.94s for the same benchmarks. These timing results

suggest that the domain operations reported in this paper are not as grossly inefficient as one might expect.

In order to assess to precision of the analysis, columns 3–6 of Figure 1 present statistics on the frequency with which extreme point elimination is required. Columns 5 and 5 give the ratio and percentages of the number of times the projection algorithm actually applies extreme point elimination and therefore (possibly) loses precision. The percentages are low (with the notable exception of ime_v2-2-1). How much precision is lost has been assessed in columns 4 and 5 which show how many variables remain to be projected out when the extreme point method takes over. Note that even at this stage, inequalities, that do not mention the variables that remain to be eliminated, are already in the projection space and are therefore exact. The ratio between the number of variables and projections approximated, indicates that typically 3 or less variables remain to be eliminated when the extreme point elimination is applied. Columns 7, 8 and 9 respectively report statistics on the way $\mathit{project}(Y, E)$ is called, namely, average $|\mathit{var}(E)|$, $|E|$ and $(\sum_{e \in E} |e|)/|E|$ where $|e|$ denotes the number variables of e with non-zero coefficients. These figures suggest that sparsity is the norm in argument-size analysis, which helps to explain the low number of calls to the extreme point algorithm. (Note that although the mean number of variables is low, one projection operation in aqua_c eliminates 60 out of 90 variables.) Interestingly, widening after one rather than two SCC iterations almost always reduces ratio of approximated projections and variables.

Finally, approximately 1% of the inequalities generated by Fourier-Motzkin projection contain very large, relatively prime coefficients (only observed for aqua_c). These inequalities often arise alongside a low coefficient inequality that almost exactly describes the same half-space. These large coefficient inequalities obfuscate the presentation of the results and slowdown the analysis with costly arbitrary-precision arithmetic. In the spirit of the weakly relational domains that use inequalities with coefficients of -1, 0 or 1 [7,25], we discard any inequality which contains a coefficient whose absolute value exceeds a preset bound. The large coefficient issue has only been observed on very large benchmarks and understanding the conditions in which it arises will be a topic for future work.

7 Related Work

Huynh, Lassez and Lassez [14] observed that sparsity is a key issue in variable elimination and suggest applying Fourier-Motzkin on (small) sparse systems and their extreme point method for dense systems. However, the context of their work was originally output in constraint logic programming [16] where over-approximation is usually unacceptable. They therefore systematically enumerate all extreme points in a breadth-first manner. Curiously, they do not consider switching between different projection strategies depending on the density of the system which, as this paper shows, is a good strategy.

Lassez, Huynh and McAloon [20] catalogue different types of redundant inequalities which include so-called syntactic and quasi-syntactic redundancies (as

discussed in Section 3.3). They identify five other classes of redundancies that reduce to syntactic and quasi-syntactic redundancies if all equalities are removed from the inequality system. For example, pairs of opposing inequalities such as $x - y \leq 5$ and $-x + y \leq -5$ can be merged into $x - y = 5$ and all occurrences of x in the remaining inequalities can be replaced by $y + 5$. Note that merging inequalities with opposing coefficients and constants does not find implicit equalities whose opposing inequalities are linear combinations of two or more inequalities. Implicit equalities can be readily detected with a Simplex solver and future work will assess whether the benefit of removing all such equalities justifies the cost of their detection.

Our implementation of Fourier-Motzkin can potentially be further refined by applying Kohler's rule [18]. Kohler distilled his observations on extreme vectors (mentioned in Section 4) into a cheap strategy to avoid generating redundant inequalities during Fourier-Motzkin elimination. The idea is to count the number of inequalities in the original system that feed into an inequality e produced in the n-th elimination step. The observation is that if the count of e exceeds $n+1$ then e is redundant. Kohler's rule has not been applied in our implementation because its correctness can, in general, be compromised when it is combined with other redundancy removal techniques [14].

8 Conclusion

This paper presented algorithms to approximate the projection and convex hull operations on the abstract domain of polyhedra, thereby providing an alternative to the classic approach based on the (potentially exponential) frame representation. Experimental results show that the sparsity of inequalities generated by program analyses allows most operations to be carried out exactly. Being able to approximate only when the size of the result becomes unmanageable is a distinct advantage over weakly relational domains which sacrifice precision up front.

Acknowledgements. We thank Jacob Howe and Peter Linnington for discussions on polyhedra and Lunjin Lu and Jonathan Martin whose work [17,23] motivated this study. This work was partly supported by EPSRC project EP/C015517.

References

1. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library. In *Static Analysis Symposium*, volume 2477 of *LNCS*, pages 213–229. Springer-Verlag, 2002.
2. F. Benoy, A. King, and F. Mesnard. Computing Convex Hulls with a Linear Solver. *Theory and Practice of Logic Programming*, 5(1&2):259–271, 2005.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Programming Language Design and Implementation*, pages 196–207. ACM Press, 2003.
4. K.-H. Borgwardt. The average number of pivot steps required by the simplex method is polynomial. *Zeitschrift für Operations Research*, 26:157–177, 1982.

5. A. Bossi, N. Cocco, and M. Fabris. Norms on Terms and their Use in Proving Universal Termination of a Logic Program. *TCS*, 124:297–328, 1994.
6. V. Chandru, C. Lassez, and J.-L. Lassez. Qualitative Theorem Proving in Linear Constraints. *Annals of Math. and Artificial Intelligence*, To appear.
7. R. Claris and J. Cortadella. The Octahedron Abstract Domain. In R. Giacobazzi, editor, *Static Analysis Symposium*, volume 3148 of *LNCS*, pages 312–327, 2004.
8. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Constraints among Variables of a Program. In *POPL*, pages 84–97. ACM Press, 1978.
9. D. De Schreye and S. Decorte. Termination of Logic Programs: The Never-Ending Story. *The Journal of Logic Programming*, 19&20:199–260, 1994.
10. D. De Schreye and K. Verschaetse. Deriving Linear Size Relations for Logic Programs by Abstract Interpretation. *New Generat. Comput.*, 13(2):117–154, 1995.
11. R. J. Duffin. On Fourier's Analysis of Linear Inequality Systems. *Mathematical Programming Study*, 1:71–95, 1974.
12. R. Giacobazzi, S. K. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *J. Logic Program.*, 3(25), 1995.
13. N. Halbwachs, D. Merchat, and C. Parent-Vigouroux. Cartesian Factoring of Polyhedra in Linear Relation Analysis. In *Static Analysis Symposium*, volume 2694 of *LNCS*. Springer Verlag, June 2003.
14. T. Huynh, C. Lassez, and J.-L. Lassez. Practical Issues on the Projection of Polyhedral Sets. *Annals of Math. and Artificial Intelligence*, 6(4):295–315, 1992.
15. J.-L. Imbert. Fourier's Elimination: Which to Choose? In *First Workshop on Principles and Practice of Constraint Programming*, pages 117–129, 1993.
16. J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. C. Yap. Output in CLP(R). In *Fifth Generation Computer Systems*, volume 2, pages 987–995, Tokyo, 1992.
17. A. King and J. C. Martin. Control Generation by Program Transformation. *Fundamenta Informaticae.* To appear.
18. D. A. Kohler. Projections of Convex Polyhedral Sets. Operations Research Centre Report ORC 67-29, University of California, Berkeley, 1967.
19. J.-L. Lassez. Querying Constraints. In *Symposium on Principles of Database Systems*, pages 288–298. ACM Press, 1990.
20. J.-L. Lassez, T. Huynh, and K. McAloon. Simplification and Elimination of Redundant Linear Arithmetic Constraints. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming*, pages 73–87. The MIT Press, 1993.
21. H. Le Verge. A Note on Chernikova's algorithm. Technical Report 1662, Institut de Recherche en Informatique, Campus Universitaire de Beaulieu, France, 1992.
22. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.
23. L. Lu and A. King. Determinacy Inference for Logic Programs. In *European Symposium on Programming*, volume 3444 of *LNCS*, pages 108–123. Springer, 2005.
24. F. Mesnard and R. Bagnara. cTI: a Constraint-Based Termination Inference Tool for ISO-Prolog. *Theory and Practice of Logic Programming*, 5(1&2):243–257, 2005.
25. A. Miné. The Octagon Abstract Domain. In *Eighth Working Conference on Reverse Engineering*, pages 310–319. IEEE Computer Society, 2001.
26. T. S. Motzkin. *Beiträge zur Theorie der Linearen Ungleichungen.* PhD thesis, Universität Zürich, 1936.
27. A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986.
28. A. Simon and A. King. Convex Hull of Planar H-Polyhedra. *International Journal of Computer Mathematics*, 81(4):259–271, March 2004.
29. A. Simon, A. King, and J. M. Howe. Two Variables per Linear Inequality as an Abstract Domain. In *Proceedings of Logic-Based Program Development and Transformation*, volume 2664 of *LNCS*, pages 71–89. Springer-Verlag, 2002.

Secure Information Flow as a Safety Problem*

Tachio Terauchi[1] and Alex Aiken[2]

[1] EECS Department, University of California, Berkeley
[2] Computer Science Department, Stanford University

Abstract. The termination insensitive secure information flow problem can be reduced to solving a safety problem via a simple program transformation. Barthe, D'Argenio, and Rezk coined the term "self-composition" to describe this reduction. This paper generalizes the self-compositional approach with a form of information downgrading recently proposed by Li and Zdancewic. We also identify a problem with applying the self-compositional approach in practice, and we present a solution to this problem that makes use of more traditional type-based approaches. The result is a framework that combines the best of both worlds, i.e., better than traditional type-based approaches and better than the self-compositional approach.

1 Introduction

A termination insensitive secure information flow problem can be defined as follows:

Definition 1 (Secure Information Flow). *Given a program P whose variables $H = \{h_1, h_2, \ldots, h_n\}$ are high security variables and $L = \{l_1, \ldots, l_n\}$ are low-security variables, P is said to be secure if and only if the values of L at the point P terminates are independent of the initial values of H.*

In this paper, we only deal with the case where programs are deterministic. The secure information flow problem is a type of non-interference problem. In practice, it expresses the problem of whether some selected information in a program or a fragment of a program (i.e., the information stored in the high-security variables) does not leak to an adversary (i.e., the low-security variables). Secure information flow has applications in software security. There is an excellent survey by Sabelfeld and Myers on issues ranging from applications to analysis techniques [1]. We note that the definition above can be extended to multi-label cases (i.e., beyond just "high" and "low") by posing the problem multiple times with different choices of high-security variables and low-security variables.

An equivalent way to state the termination insensitive secure information flow problem is:

* This research was supported in part by NASA Grant No. NNA04CI57A; NSF Grant Nos. CCR-0234689, CCR-0085949, and CCR-0326577. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 352–367, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Definition 2 (Secure Information Flow - Alternative Definition). *Given a program P whose variables $H = \{h_1, \ldots, h_n\}$ are high security variables and $L = \{l_1, \ldots, l_n\}$ are low-security variables, P is said to be secure if and only if for any stores M_1 and M_2 such that $M_1|_{H^c} = M_2|_{H^c}$,*

$$(\langle M_1, P\rangle \neq \bot \wedge \langle M_2, P\rangle \neq \bot) \Rightarrow \langle M_1, P\rangle|_L = \langle M_2, P\rangle|_L$$

Formally, a store M is a mapping from variables to values. The notation $M|_X$ is the restriction of the store M to the variable domain X, i.e., $M|_X = \{x \mapsto v \mid (x \mapsto v) \in M \wedge x \in X\}$. The set X^c is the complement of X. If P terminates given the initial store M, $\langle M, P\rangle$ denotes the final store; $\langle M, P\rangle = \bot$ if non-terminating.

Both definitions appear frequently with some variation in superficial details. It is easy to see that the definitions are equivalent. The second definition is particularly nice for our purpose because it is easy to see the reduction from the definition into a safety problem. Intuitively, a safety property is a property of a program which can be refuted by observing a finite trace of the program. Our definition of secure information flow only concerns the final store. Then a safety problem can be formally defined as

Definition 3 (Safety). *Let Pr be the set of all programs (for some fixed programming language). Then a safety property is a set $S \subseteq Pr$ such that there exists a logical formula $\phi(X, Y)$ such that*

$$S = \{P \mid \forall M. \langle M, P\rangle \neq \bot \Rightarrow \phi(\langle M, P\rangle, M)\}$$

A *safety problem* is a membership problem for some safety property.

Secure information flow, termination sensitive or not, is not a safety property (see, e.g., [2] for a proof). However, the termination insensitive secure information flow problem is *almost* a safety problem. To this end, we introduce the concept of a 2-*safety property* which is intuitively a property that can be refuted by observing *two* finite traces. More formally,

Definition 4 (2-Safety). *Let Pr be the set of all programs (for some fixed programming language). Then a 2-safety property is a set $S \subseteq Pr$ such that there exists a logical formula $\phi(X, Y, Z, W)$ such that*

$$S = \{P \mid \forall M_1, M_2. (\langle M_1, P\rangle \neq \bot \wedge \langle M_2, P\rangle \neq \bot) \Rightarrow \phi(\langle M_1, P\rangle, \langle M_2, P\rangle, M_1, M_2)\}$$

To distinguish, we say 1-safety when we mean safety. Clearly, any 1-safety property is a 2-safety property. The following is immediate:

Theorem 1. *The termination insensitive secure information flow problem is a 2-safety problem.*

For any program P, let $V(P)$ be the set of all variables appearing in P and let $C(P)$ be the copy of P with each $x \in V(P)$ replaced by a fresh variable $C(x)$. Any 2-safety problem can be reduced to a 1-safety problem by the following *self-composition* reduction:

Definition 5 (Self-composition). *Let S be a 2-safety property, i.e., $S = \{P \mid \forall M_1, M_2.(\langle M_1, e\rangle \neq \bot \wedge \langle M_2, e\rangle \neq \bot) \Rightarrow \phi(\langle M_1, e\rangle, \langle M_2, e\rangle, M_1, M_2)\}$ for some ϕ. Then a self-composition reduction of S is the set*

$$\{P' \mid P' = P; C(P) \wedge \forall M_1, M_2.\langle M_1 \cup C(M_2), P'\rangle \neq \bot \Rightarrow \theta\}$$

where $\theta = \phi(\langle M_1 \cup C(M_2), P'\rangle|_{V(P)}, \langle M_1 \cup C(M_2), P'\rangle|_{V(C(P))}, M_1, M_2)$.

where the symbol ; is the sequential composition. It is easy to see that a self-composition of any 2-safety property S is a recursive subset of some 1-safety property S', i.e., given an oracle access to S', we can decide (in fact easily) if $P \in S''$ where S'' is the self-composition reduction of S. Furthermore it is easy to see that the self-composed form is equivalent to the original in the following sense:

Theorem 2. *Let S be a 2-safety property and let S' be its self-composition. Then $P \in S$ if and only if $P; C(P) \in S'$.*

Thus any 2-safety problem can be solved by reducing it to an equivalent 1-safety problem via self-composition and then solving the 1-safety problem.

In the case of the termination insensitive secure information flow problem, self-composition reduces the problem into the following problem:

Definition 6 (Secure Information Flow - Self-composed Version). *Given a program P whose variables $H = \{h_1, \ldots, h_n\}$ are high security variables and $L = \{l_1, \ldots, l_n\}$ are low-security variables, P is said to be secure if and only if for any stores M_1 and M_2 such that $dom(M_1) = V(P)$ and $dom(M_2) = V(C(P))$ and $M_1|_{H^c} = M_2|_{C(H^c)}$,*

$$\langle M_1 \cup M_2, P; C(P)\rangle \neq \bot \Rightarrow C(\langle M_1 \cup M_2, P; C(P)\rangle|_L) = \langle M_1 \cup M_2, P; C(P)\rangle|_{C(L)}$$

where $C(M)$ is a store identical to M except that each variable x appearing in M is replaced by $C(x)$. Note that it is possible to see the above formulation directly from Definition 2 without going through the generalization of defining a 2-safety property as we have done here. As far as we know, the direct formulation appears in at least two recent papers [3,4]. We borrowed the term "self-composition" from Barthe, D'Argenio, and Rezk [4], although they define it slightly differently.

Self-composition is a promising approach to solving difficult secure information flow instances thanks to the recent success on generic automatic software safety analysis tools such as SLAM [5] and BLAST [6], to name a few. Both SLAM and BLAST combine theorem proving and model checking in an iteratively refining manner to achieve robust safety analysis that can scale to programs of non-trivial size written in feature-rich programming languages like C. Also, they are in theory *almost* complete [7]. In practice, they have been able to verify many safety properties that were too difficult for older approaches that were not fully path-sensitive and sometimes not even flow-sensitive.

```
z := 1;
if (h) then x := 1 else skip;
if (¬h) then x := z else skip;
l := x + y
```

Fig. 1. The variable h is high-security and the variable l is low-security. (The variable y is not high-security.) This code is secure: regardless of the valuation of h, the low-security variable l will be $1 + y$ at the end of the program.

```
z := 1;
if (h) then x := 1 else skip;
if (¬h) then x := z else skip;
l := x + y;
z' := 1;
if (h') then x' := 1 else skip;
if (¬h') then x' := z' else skip
l' := x' + y'
```

Fig. 2. Self-composition reduction applied to the program in Figure 1. For each variable x, $C(x) = x'$.

What does this progress in automatic safety analysis actually mean to secure information flow? For example, type-based information flow analysis algorithms, flow-sensitive or not, cannot show that the program shown in Figure 1 is secure since the low-security variable l is assigned in a branch of a conditional that depends on the high-security variable h. But a self-compositional approach can easily check that this program is secure as follows. Figure 2 is the result of applying the self-composition reduction to the program. The safety problem of whether $l = l'$ at the end of the program given $x = x' \wedge y = y' \wedge z = z' \wedge l = l'$ at the entry can be verified easily by a modern safety analysis tool. So by Theorem 2, we have automatically proved that the original program is secure. In fact, Theorem 2 implies that given a complete safety analysis, we can solve the termination insensitive secure information flow problem completely.

Before we go on to the main results of the paper, we note that it is fairly easy to carry out a similar construction for termination *sensitive* secure information flow problem by defining a "2-liveness" property which may observe up to two possibly *infinite* traces to refute the property. Self-composition can then be defined using a parallel composition instead of a sequential composition to reduce any 2-liveness problem to a 1-liveness problem. But since there are not practical frameworks for checking general software liveness properties (though some promising proposals are starting to appear [8]), we limit the content of this paper to the termination insensitive case. Also, non-deterministic programs are outside of the scope of this paper.

1.1 Contributions

The two main contributions of this paper are as follows:

- We extend the self-compositional approach to the secure information flow problem with information downgrading recently proposed by Li and Zdancewic [9].
- We identify a problem with applying the self-compositional approach in practice. We then present a solution to this problem that makes use of more traditional type-based approaches.

The first contribution was motivated by an elegant characterization of information downgrading called *relaxed non-interference* proposed recently by Li and Zdancewic [9]. Their paper contains a type-based approach for automatically checking relaxed non-interference. The self-compositional approach can in theory verify a wider range of secure programs than their type-based approach.

The second contribution starts from a disappointing discovery that the self-compositional approach, even when combined with current state-of-the-art generic automatic safety analysis tools, is too inefficient in practice. We will point out why this is the case, and offer a remedy based on previous and ongoing research on type-based approaches to secure information flow, including Li and Zdancewic type system for information downgrading. The result is a framework that combines the best of both worlds, i.e., better than type-based approaches and better than the self-compositional approach.

2 Information Downgrading

"Vanilla" secure information flow as defined in Section 1 is often criticized for being too strict. For example, a security policy may permit information stored in the high-security variable *secret* to leak as long as the hash of the password from the user, say initially stored in the non-high-security variable *input*, matches with the high-security variable *hash*. For example, the following program is secure according to this policy:

$$\texttt{if } (hashfunc(input) = hash) \texttt{ then } l \texttt{ := } secret \texttt{ else skip;}$$

where l is a low-security variable. Unfortunately, the above program is not secure according to the definition of vanilla secure information flow because the valuation of l depends on the valuation of the high-security variable *secret* (and on *hash* too). In general, vanilla secure information flow does not allow *any* method of leaking *anything* about the high-security variables.

Researchers have proposed various ways to relax secure information flow to permit policies like the one above, such as robust declassification [10], delimited information release [11], and abstract non-interference [12]. A particularly nice approach called *relaxed non-interference* has been recently proposed by Li and Zdancewic [9]. Their idea is to express downgrading by the existence of a *clean function* that takes "downgraded" high-security information but does not look directly at high-security variables. Their paper is restricted to the purely functional

setting, but when extended to the imperative setting, their idea can be described roughly as follows. A security policy is stated by associating each high-security variable h_i to a *downgrading function* f_{h_i}, and then we define the security of a program P by the existence of a program $F(f_{h_1}(h_1), \ldots, f_{h_n}(h_n))$ such that F does not mention the high security variables and $F(f_{h_1}(h_1), \ldots, f_{h_n}(h_n))$ agrees with P on low-security variables at termination. Here, the notation $F(e_1, \ldots, e_n)$ refers to a program that first evaluates $e_1, \ldots, e_n$ and stores them in some variables prior to the evaluation of the rest of the program. $F(e_1, \ldots, e_n)$ can be arbitrary powerful, i.e., it need not be computable. (Readers familiar with relaxed non-interference may notice another difference – in addition to the imperative extension – from Li and Zdancewic's original definition, i.e., the use of semantic equivalence instead of syntactic equivalence rules. The consequence of this difference is discussed later in this section.) Note that secure information flow with information downgrading is more general than vanilla secure information flow; vanilla secure information flow can be expressed by setting all downgrading functions to the constant function $\lambda x.0$.

For example, in our password example, the downgrading function for *secret* can be set to

$$f = \lambda x.\texttt{if}\ (\mathit{hashfunc}(\mathit{input}) = \mathit{hash})\ \texttt{then}\ x\ \texttt{else}\ c$$

where c is some constant not in the range of values for *secret*. Then, one only needs to prove that there exists F such that $F(f(\mathit{secret}))$ is equivalent to our original program, which in this case is true by inspection. Relaxed non-interference is surprisingly general and natural. For example, it is easy to see that associating the downgrading function $\lambda x.\mathit{length}(x)$ to a secret string data implies that only the length of the string may be leaked.

We simplify the definition slightly for purpose of exposition. Formally, we use the following definition of the terminating insensitive secure information flow with information downgrading.

Definition 7 (Secure Information Flow with Information Downgrading). *Given a program P whose variables $H = \{h_1, \ldots, h_n\}$ are high security variables and $L = \{l_1, \ldots, l_n\}$ are low-security variables, P is said to be secure with respect to the downgrading policy e if and only if there exists F such that F does not mention any variable in H and for any M,*

$$\langle M, P\rangle \neq \bot \Rightarrow (\langle M, F(e)\rangle \neq \bot \wedge \langle M, P\rangle|_L = \langle M, F(e)\rangle|_L)$$

Here, e is any side-effect free expression. It is easy to see that our definition is at least as expressive as Li and Zdancewic style of using explicit downgrading functions. For example, vanilla secure information flow can be obtained by setting e to be the tuple $(h_1, \ldots, h_n)$. For the password example, e is

$$\texttt{if}\ (\mathit{hashfunc}(\mathit{input}) = \mathit{hash})\ \texttt{then}\ \mathit{secret}\ \texttt{else}\ c$$

It is worth pointing out that the above definition is slightly different from that of Li and Zdancewic's since we use semantic equivalence to check that

$\langle M, P\rangle|_L = \langle M, F(e)\rangle|_L$ whereas Li and Zdancewic take a less complete (but still sound) equivalence relation as the definition. Their paper contains a discussion on why a weaker equivalence may be desirable in some situations. However, it is not clear whether using a weaker equivalence based on intentional syntactic equivalence rules as done in their paper is best. Perhaps a more principled approach is to equate some computational hardness properties as well as semantic equivalence. For example, any $F(\lambda x.\mathtt{if}\ (\mathit{password} = x)\ \mathtt{then}\ 1\ \mathtt{else}\ 0)$ semantically equivalent to $l := \mathit{password}$ on the variable l will be computationally expensive assuming that the set of valuations of *password* is large. Note that there is an F such that semantic equivalence alone will not be able to distinguish $F(\lambda x.\mathtt{if}\ (\mathit{password} = x)\ \mathtt{then}\ 1\ \mathtt{else}\ 0)$ from $l := \mathit{password}$, namely the one that tries all possible strings. In this paper, we stick with semantic equivalence.

We now prove the following.

Theorem 3. *The termination insensitive secure information flow with information downgrading is a 2-safety problem.*

The formal proof appears in our companion technical report [13]. The proof establishes the equivalence of Definition 7 to the following predicate

$$\forall M_1, M_2.(\langle M_1, P\rangle \neq \bot \wedge \langle M_2, P\rangle \neq \bot) \Rightarrow$$
$$((M_1|_{H^c} = M_2|_{H^c} \wedge \langle M_1, e\rangle = \langle M_2, e\rangle) \Rightarrow \langle M_1, P\rangle|_L = \langle M_2, P\rangle|_L)$$

The predicate is actually equivalent to the definition of *delimited information release* [11] restricted to the safety case. Therefore, the above proof shows that relaxed non-interference with semantic equivalence is roughly (modulo the imperative extension) equivalent to that of delimited information release. Since Barthe, D'Argenio, and Rezk [4]'s formulation of self-composition is flexible enough to handle delimited information release, our result also shows that their framework can be used as a black box to solve secure information flow problems with information downgrading in the style of relaxed non-interference.

Concretely, self-composition reduces the termination insensitive secure information flow with information downgrading to the following problem:

Definition 8 (Secure Information Flow with Information Downgrading - Self-composed Version). *Given a program P whose variables $H = \{h_1, \ldots, h_n\}$ are high security variables and $L = \{l_1, \ldots, l_n\}$ are low-security variables, P is said to be secure with respect to the downgrading policy e if and only if for any stores M_1 and M_2 such that $dom(M_1) = V(P)$ and $dom(M_2) = V(C(P))$, $M_1|_{H^c} = M_2|_{C(H^c)}$, and $\langle M_1, e\rangle = \langle M_2, C(e)\rangle$,*

$$\langle M_1 \cup M_2, P; C(P)\rangle \neq \bot \Rightarrow C(\langle M_1 \cup M_2, P; C(P)\rangle|_L) = \langle M_1 \cup M_2, P; C(P)\rangle|_{C(L)}$$

As in the case of vanilla secure information flow, this self-compositional reduction is complete. Hence in theory, a complete safety analysis can decide any instance of the problem. In practice, the self-compositional approach can check cases where Li and Zdancewic's type-based approach would fail. For example, the program in Figure 3 is secure according the downgrading policy

```
if (hashfunc(input) = hash) then
   t := t + 1; l := l + secret
else skip
```

Fig. 3. The variables *secret* and *hash* are high-security and the variable *input* and l are low-security. This code is secure according to the downgrading policy `if` $(hashfunc(input) = hash)$ `then` *secret* `else` 0.

`if` $(hashfunc(input) = hash)$ `then` *secret* `else` c. Essentially, the program is same as our original example except that we have added a few small things so that the code isn't exactly like the downgrading policy. The program can be easily proved to be secure via the self-compositional approach; the downgrading policy leads to a conditional predicate, but that is no harder than handling conditionals in the program body, and therefore a path-sensitive safety analysis can quickly check that the safety property is satisfied in the self-composed program (not shown). On the other hand, conventional type-based approaches would break in the presence of these small changes since they are more dependent on the structure of downgrading operations.

3 Self-composition in Practice, Its Problem, and a Solution

The main appeal of the self-compositional approach to secure information flow comes from the recent successes with automatic safety analysis tools in verifying a very broad range of safety properties in real programs, including ones that are path-sensitive, flow-sensitive, and (linear) arithmetic sensitive. Furthermore, automatic safety property checking is an active area of research with frequent improvements, and therefore even if some self-composed instances of a secure information flow problem cannot be solved by the existing tools today, it may not be unreasonable to expect them to be solved by the next generation of safety analysis tools. That is, the self-compositional approach automatically benefits from improvements to the underlying safety analysis. Furthermore, the self-compositional approach needs nothing more than off-the-shelf tools, and so it has an engineering advantage over type-based approaches.

In this section, we argue that such an optimistic prospect is unrealistic in practice. When we actually applied the self-composition approach, we found that not only are the existing automatic safety analysis tools not powerful enough to verify many realistic problem instances efficiently (or at all), but also that there are strong reasons to believe that it is unlikely to expect any future advance in safety analysis designed for "natural" safety problems (i.e., ones that are naturally 1-safety) to be able to close the gap significantly.

We first motivate our argument by a simple example. Figure 4 is a program which computes the nth Fibonacci number and sets the low-security variable l to 1 if the nth Fibonacci number is greater than k and to 0 otherwise. The program

```
while (n > 0) do
   f1 := f1 + f2; f2 := f1 - f2; n := n - 1;
if (f1 > k) then l := 1 else l := 0;
```

Fig. 4. The while loop computes the nth Fibonacci number. The variable l is low security, which is set to 1 if the nth Fibonacci number is greater than k, and is set to 0 otherwise. There are no high-security variables.

```
while (n > 0) do
   f1 := f1 + f2; f2 := f1 - f2; n := n - 1;
if (f1 > k) then l := 1 else l := 0;
while (n' > 0) do
   f1' := f1' + f2'; f2' := f1' - f2'; n' := n' - 1;
if (f1' > k') then l' := 1 else l' := 0;
```

Fig. 5. The program in Figure 4 after self-composition

contains no high-security variables, so it is trivially secure. Let us apply the self-composition reduction by renaming each variable x to x' in the copy (shown in Figure 5). We would like the safety analysis tools to check that $l = l'$ at the end of the program provided that for each variable x in the original, $x = x'$ at the beginning of the program. However, a state-of-the-art safety analysis tool BLAST [6] fails to terminate given this query; more precisely, BLAST endlessly keeps discovering more and more predicates getting closer and closer to the answer but never actually converging.[1]

Why does this happen? The reason is that the modern generic safety analysis tools gain their robustness by moving away from structure-dependent reasoning and instead trying to solve the problem semantically. In the case above, if BLAST could verify that $l = l'$ at the end of the self-composed program, then that roughly means that it was able to show that the upper part of the original code was computing a Fibonacci number for each n. We believe that this problem also applies to other safety analysis tools for imperative languages based on a Hoare-style reasoning framework since the framework encourages verifying a property about the whole program by locally reasoning about the store update at each statement. We give more details supporting this argument in Section 3.1.

Even if BLAST was improved with more arithmetic-related reasoning power or if we used another tool that can verify the correctness of our Fibonacci computation loop, there would be always another example whose partial correctness would be too difficult for the tool to verify automatically. Why does this matter to the self-compositional approach to secure information flow? Because there are many programs that compute arbitrary values in complex ways, and it is fair to expect that these values can flow to low-security variables since the low-security

[1] We used the latest version (as of March 2005) obtained directly from the BLAST group.

$$P ::= x \texttt{ := } e \mid \texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \mid \texttt{while } e \texttt{ do } P \mid P_1; P_2 \mid \texttt{skip}$$

Fig. 6. The syntax of `While`. e is some reasonable expression such as integer arithmetics, comparisons, and boolean operations.

$$\varepsilon ::= [\,] \mid x \texttt{ := } \varepsilon \mid \texttt{if } \varepsilon \texttt{ then } P_1 \texttt{ else } P_2 \mid \texttt{if } e \texttt{ then } \varepsilon \texttt{ else } P \mid \texttt{if } e \texttt{ then } P \texttt{ else } \varepsilon \mid \texttt{while } \varepsilon \texttt{ do } P \mid \texttt{while } e \texttt{ do } \varepsilon \mid \varepsilon; P \mid P; \varepsilon$$

Fig. 7. The contexts of `While`

variables are the observable outputs of the program. (On the other hand, parts of the program where high-security values flow can be expected to be small and not too complex in most real security-aware applications.)

Therefore, what the self-compositional approach needs is some reasoning extension that can make use of the inherent symmetry and redundancy in self-composed programs but not in ordinary programs. For example, in the case of the Fibonacci program, this reasoning extension should be able to tell that the loops are equivalent by the fact that both loops are just copies of the same code with each copied variable in the code starting with the same value as the original. On the other hand, if copies of some code actually use variables with different initial values, then this reasoning system should safely say that "I do not know if they are equivalent" so that a more powerful reasoning logic can work out the details.

Such a reasoning extension is exactly where type-based approaches to secure information flow excel. That is, the "same value variables" are the low security variables, and "different value variables" are the high-security variables. Indeed, type-based approaches can easily verify our Fibonacci program by carrying out roughly the following logical reasoning: f_1 is only assigned low-security values in a while loop with a low-security guard, and hence l is assigned only in a conditional statement of a low-security condition which implies that l is low-security. But as we have seen in the previous sections, there are instances of secure information flow that cannot be verified by type-based approaches but can be easily verified by the self-compositional approach. To this end, we generalize this line of thought to design an approach to secure information flow that combines the best parts of the two approaches.

3.1 Type-Directed Transformation for Secure Information Flow

We illustrate our idea using the imperative language `While` defined in Figure 6. The semantics of `While` is completely standard. While we choose this simple language for purpose of exposition, it is not hard to adapt our approach to more complex languages.

$$\frac{\Gamma \vdash e : \tau \text{ where } \tau \text{ is a low-security type}}{x \texttt{ := } e \rightarrow_\Gamma x \texttt{ := } e; C(x) \texttt{ := } x}$$

$$\frac{\Gamma \not\vdash e : \tau \text{ where } \tau \text{ is a low-security type}}{x \texttt{ := } e \rightarrow_\Gamma x \texttt{ := } e; C(x) \texttt{ := } C(e)}$$

$$\frac{\Gamma \vdash e : \tau \text{ where } \tau \text{ is a low-security type} \quad P_1 \rightarrow_\Gamma P_1^* \quad P_2 \rightarrow_\Gamma P_2^*}{\texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \rightarrow_\Gamma \texttt{if } e \texttt{ then } P_1^* \texttt{ else } P_2^*}$$

$$\frac{\Gamma \not\vdash e : \tau \text{ where } \tau \text{ is a low-security type}}{\texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \rightarrow_\Gamma \texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2; \texttt{if } C(e) \texttt{ then } C(P_1) \texttt{ else } C(P_2)}$$

$$\frac{\Gamma \vdash e : \tau \text{ where } \tau \text{ is a low-security type} \quad P \rightarrow_\Gamma P^*}{\texttt{while } e \texttt{ do } s \rightarrow_\Gamma \texttt{while } e \texttt{ do } P^*}$$

$$\frac{\Gamma \not\vdash e : \tau \text{ where } \tau \text{ is a low-security type}}{\texttt{while } e \texttt{ do } P \rightarrow_\Gamma \texttt{while } e \texttt{ do } P; \texttt{while } C(e) \texttt{ do } C(P)}$$

$$\frac{P_1 \rightarrow_\Gamma P_1^* \quad P_2 \rightarrow_\Gamma P_2^*}{P_1; P_2 \rightarrow_\Gamma P_1^*; P_2^*} \qquad \frac{}{\texttt{skip} \rightarrow_\Gamma \texttt{skip}}$$

Fig. 8. Type-directed translation $\rightarrow_\Gamma$. "$\Gamma \not\vdash e : \tau$ where τ is a low-security type" means that $\Gamma \vdash e : \tau$ is not derivable for any low-security type τ.

To motivate the idea, consider the program $P = \texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2$. If a secure information flow type system gives e a low-security type, then the self composition $P; C(P)$ is equivalent to the program $\texttt{if } e \texttt{ then } (P_1; C(P_1)) \texttt{ else } (P_2; C(P_2))$ provided that the values of the low-security variables between the original and the copy are equal at the beginning of the program. Now, suppose that e is (or was the result of) a complex computation like our Fibonacci loop. Then using the second form instead of $P; C(P)$, a safety analysis tool is able to bypass checking whether e is equal to $C(e)$ without losing precision or efficiency. Furthermore, we may recursively apply the same idea to P_1 and P_2 so that we may not even need to use $C(P_1)$ and $C(P_2)$.

We now generalize this idea to design a *type-directed transformation* for secure information flow. To this end, we first define the contexts ε of `While` in a completely standard manner given in Figure 7. Our type-directed transformation is parametrized by a secure information flow type system. Rather than defining a type-directed transformation for each different type system and proving the correctness each time, we formally state what our type-directed transformation expects from a secure information flow type system so that we can design one type-directed transformation for all type systems satisfying the definition and prove its correctness once and for all.

Definition 9. *Given a secure information flow problem with information downgrading problem instance (P, H, L, e) (see Definition 7), secure information flow*

type inference is an algorithm that outputs a type environment Γ with the relation $\sim_\Gamma$ satisfying all of the following.

(1) For any M_1 and M_2, if $M_1|_{H^c} = M_2|_{H^c}$ and $\langle M_1, e\rangle = \langle M_2, e\rangle$ then $M_1 \sim_\Gamma M_2$.
(2) For any P such that $\Gamma \vdash P$ and for any M_1 and M_2 such that $M_1 \sim_\Gamma M_2$, $\langle M_1, P\rangle \sim_\Gamma \langle M_2, P\rangle$.
(3) For any e such that $\Gamma \vdash e : \tau$ and τ is a low-security type, for any M_1 and M_2 such that $M_1 \sim_\Gamma M_2$, $\langle M_1, e\rangle = \langle M_2, e\rangle$.
(4) For any ε and P, if $\Gamma \vdash \varepsilon[P]$, then $\Gamma \vdash P$.
(5) $\Gamma \vdash P$

Intuitively, the first condition says that the precondition of the original security policy is at least as strong as the relation $\sim_\Gamma$. The second condition says that $\sim_\Gamma$ is preserved by the program semantics. The third condition says that if an expression is typed with a low-security type, then it in fact is low-security with respect to $\sim_\Gamma$. The fourth condition is a standard structural property for (flow-insensitive) type systems. The last condition says that P itself can be typed under Γ.

For example, the well-known Volpano and Smith type inference algorithm [14] when restricted to the language `While` can satisfy the above requirement for vanilla secure information flow (i.e., the downgrading policy e is some constant) by letting

$$\sim_\Gamma = \{(M_1, M_2) \mid M_1(x) = M_2(x), x{:}\tau \in \Gamma \text{ where } \tau \text{ is a low-security type}\}$$

Defining $\sim_\Gamma$ for Li and Zdancewic type system [9] (when adapted to the language `While` in a natural way) is also not difficult:

$$\sim_\Gamma = \{(M_1, M_2) \mid \langle M_1, e\rangle = \langle M_2, e\rangle, \Gamma \vdash e : \tau \text{ where } \tau \text{ is a low-security type}\}$$

(Indeed, this definition, also works for the Volpano and Smith type system although it is unnecessarily more elaborate than the one above. This fact is not surprising since Li and Zdancewic system can be thought of as an extension to the Volpano and Smith system.) Due to space constraints, we do not formally describe any specific type inference algorithm in this paper and instead ask readers to refer to the cited references. Our companion technical report discusses how to adapt our approach to secure information flow type systems that do not quite meet these requirements [13].

It is important to note that we do not need an algorithm that actually computes the relation $\sim_\Gamma$. Instead, merely the existence of such a relation is enough since $\sim_\Gamma$ is only used explicitly when proving the correctness of the type-directed transformation.

We now describe our type-directed transformation. Given a problem instance (P, H, L, e) and Γ produced by the corresponding secure information flow type inference, the type-directed transformation $\rightarrow_\Gamma$ is defined by the rules shown in Figure 8. In order to solve the given problem instance, we first apply this

```
while (n > 0) do
  f1 := f1 + f2; f2 := f1 - f2; n := n - 1;
if (h) then x := 1 else skip;
if (¬h) then x := 1 else skip;
while (i < f1) do
  l := l + x; i := i + 1
```

Fig. 9. The variable h is high-security and the variable l is low-security. The program is secure but cannot be verified by either a type-based approach or self-composition.

```
while (n > 0) do
  f1 := f1 + f2; f1' := f1; f2 := f1 - f2; f2' := f2;
  n := n - 1; n' := n;
if (h) then x := 1 else skip; if (h') then x' := 1 else skip;
if (¬h) then x := 1 else skip; if (¬h') then x' := 1 else skip;
while (i < f1) do
  l := l + x; l' := l' + x'; i := i + 1; i' := i
```

Fig. 10. The program from Figure 9 after the type-directed transformation

transformation to P to obtain a program P^*, i.e., $P \rightarrow_\Gamma P^*$. Then we ask a safety analysis tool whether for any M_1 and M_2 such that $dom(M_1) = V(P)$, $dom(M_2) = V(C(P))$, $M_1|_{H^c} = M_2|_{C(H^c)}$, and $\langle M_1, e\rangle = \langle M_2, C(e)\rangle$, whether

$$\langle M_1 \cup M_2, P^*\rangle \neq \bot \Rightarrow C(\langle M_1 \cup M_2, P^*\rangle|_L) = \langle M_1 \cup M_2, P^*\rangle|_{C(L)}$$

That is, we ask the same query as the self-compositional approach except that we use P^* in place of $P; C(P)$.

As an example, consider the program shown in Figure 9. The program exhibits interactions of features discussed in previous sections that made type-based approaches and the self-composition approach fail (at least when using BLAST as the underlying safety analysis). Therefore, it can be checked by neither method. Applying the type-directed transformation using Volpano and Smith type inference algorithm, we obtain the program P^* shown in Figure 10. Note that both loop conditions remain unduplicated (though their bodies are duplicated) since both conditions can be given low-security types. BLAST can easily decide that $l = l'$ at the end of P^* provided that $n = n'$, $f_1 = f_1'$, $f_2 = f_2'$, $i = i'$, and $l = l'$ at the beginning, i.e., it can prove that the program is secure. In fact, BLAST is clever enough that it will not even bother to look carefully at the first loop (which was the part that made BLAST fail in the self-composition approach!) since it quickly notices simply by looking at the code following the loop that it can prove $l = l'$ at the end of the program regardless of what values are stored in f_1, f_1', f_2', n, and n' after the loop.

We now prove the correctness of the type-directed transformation approach. The following lemma is the main technical result.

Lemma 1. *Suppose $P \rightarrow_\Gamma P^*$ where Γ is the output of a secure information flow type system given (P, H, L, e) satisfying Definition 9. Then, for any M_1 and M_2 such that $M_1 \sim_\Gamma M_2$, if $\langle M_1, P\rangle \neq \bot$ and $\langle M_2, P\rangle \neq \bot$ then*

$$\langle M_1, P\rangle = \langle M_1 \cup C(M_2), P^*\rangle|_{V(P)} \wedge C(\langle M_2, P\rangle) = \langle M_1 \cup C(M_2), P^*\rangle|_{V(C(P))}$$

The proof appears in our companion technical report [13].

Theorem 4. *For any M_1 and M_2 such that $M_1|_{H^c} = M_2|_{H^c}$, $\langle M_1, e\rangle = \langle M_2, e\rangle$, $\langle M_1, P\rangle \neq \bot$, and $\langle M_2, P\rangle \neq \bot$*

$$\langle M_1, P\rangle|_L = \langle M_2, P\rangle|_{C(L)} \longleftrightarrow C(\langle M_1 \cup C(M_2), P^*\rangle|_L) = \langle M_1 \cup C(M_2), P^*\rangle|_{C(L)}$$

where $P \rightarrow_\Gamma P^*$ and Γ is the output of a secure information flow type system given (P, H, L, e) satisfying Definition 9.

Proof. Immediate from condition (1) in Definition 9 and Lemma 1.

Therefore the type-directed transformation approach is sound and complete up to the soundness and completeness of the underlying safety analysis.

The type-directed transformation is inexpensive relative to the complexity of the underlying type inference algorithm. It is easy to see that for $P \rightarrow_\Gamma P^*$, the size of P^* is at most two times the size of P. Computing P^* from P takes time linear in P and the number of $\Gamma \vdash e : \tau$ queries made to the type inference algorithm. However, most secure information flow type systems actually compute the principal types for each expression. In such a case, asking whether there is a low-security type τ such that $\Gamma \vdash e : \tau$ is a constant time operation once the principal types have been computed for P.

It is clear that the type-directed transformation approach is better than a type-based approach alone since it runs the type inference algorithm as a sub-process, and therefore it may accept the program if the type inference successfully assigned low-security types to the low-security variables.

Before we argue that the type-directed transformation approach is better than the self-compositional approach, we point out that in their full generality, the two approaches are equivalent since they are both a complete characterization of the same secure information flow problem, i.e., they are no different to a hypothetical safety analysis having infinite deduction power. Even restricted to the class of safety analysis tools that are "fast" and sound (but not necessarily complete), we cannot compare the two because, for example, this class includes one that rejects all programs not of the form $P; C(P)$, i.e., the self-composition approach is always better for such a safety analysis, and conversely, there is a sound safety analysis that rejects all programs of the form $P; C(P)$.

Instead, we argue that type-directed-transformed programs tend to be more digestible than self-composed programs for most automatic safety analysis tools assuming that they are targeted toward the general class of "natural" safety (i.e., naturally 1-safety) problems for imperative languages. Such tools typically reason about a program by interpreting each program statement as an abstract

store update operation where an abstract store may be a set of abstract values stored in abstract memory cells, a set of predicates over program variables where each predicate represents a possible store, or something similar. With self-composition, the store space for the copies P and $C(P)$ are completely disjoint. However, the query is all about connecting these two stores, i.e., it is about whether some portion of the two disjoint store spaces is equivalent after the program terminates given that some portion of the two disjoint store spaces is equivalent before the program. Therefore 1-safety analysis tools generally suffer from not being able to relate the two stores within the abstract interpretation phase. Our type-directed transformation directly makes *relevant* connections between the two stores locally within the program. These connections help the safety analysis significantly in some situations as seen in the example in this section (Figure 9, 10) where the self-compositional approach would perform poorly.

4 Related Work

Darvas, Hähnle, and Sands [3] used a self-compositional approach to prove secure information flow properties for Java CARD programs. They used an interactive approach instead of an automatic approach. Barthe, D'Argenio, and Rezk coined the term "self-composition" in their paper [4]. Their paper is mostly theoretical results on characterizing various secure information flow problems, including non-deterministic and termination-sensitive cases, in a self-compositional framework. We believe that our paper is the first one to examine applying an automatic safety analysis in the self-compositional setting.

Barthe, D'Argenio, and Rezk in the same paper showed that their self-compositional framework can handle delimited information release as originally proposed by Sabelfeld and Myers [11]. We have shown that Li and Zdancewic's recently proposed relaxed non-interference [9] is equivalent to delimited information release when strengthened with semantic equivalence. Relaxed non-interference is arguably a more natural formulation of information downgrading than delimited information release. Our paper suggests a promising practical approach toward making complete use of properties definable as relaxed non-interference.

5 Conclusions and Future Work

We have shown that Li and Zdancewic's relaxed non-interference can be incorporated into both self-composition and its generalization, the type-directed transformation approach. We have presented the type-directed transformation approach as a solution to a problem with applying self-composition in practice with off-the-shelf automatic safety analysis tools. The type-directed transformation approach combines the best parts of traditional type-based approaches and self-composition.

One possible improvement to our type-directed transformation is to make it iterative, i.e., in the event that the safety analysis fails, instead of failing the

whole process completely it may report back to the type system with information about which expressions are low-security at which program points. Then the type system can "cast" these expressions to low-security types to obtain more low-security expressions, and the process repeats. To make this work, we need a way to obtain partial results from the safety analysis tool. Obtaining useful partial results may be difficult for a demand-driven framework such as BLAST.

References

1. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Selected Areas in Communications **21** (2003) 5–19
2. McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: SP '94: Proceedings of the 1994 IEEE Symposium on Security and Privacy, Washington, DC, USA, IEEE Computer Society (1994) 79
3. Darvas Á, Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In Gorrieri, R., ed.: Workshop on Issues in the Theory of Security, WITS, IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS (2003)
4. Barthe, G., D'Argenio, P., Rezk, T.: Secure information flow by self-composition. In: Computer Security Fundation Workshop (CSFW'17), IEEE Press (2004)
5. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon (2002) 1–3
6. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon (2002) 58–70
7. Ball, T., Podelski, A., Rajamani, S.K.: Relative completeness of abstraction refinement for software model checking. In Kaoen, J.P., Stevens, P., eds.: Proceedings of TACAS02: Tools and Algorithms for the Construction and Analysis of Systems. Volume 2280 of LNCS., Springer-Verlag (2002) 158–172
8. Podelski, A., Rybalchenko, A.: Transition predicate abstraction and fair termination. In: Proceedings of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Long Beach, California (2005) 132–144
9. Li, P., Zdancewic, S.: Downgrading policies and relaxed noninterference. In: Proceedings of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Long Beach, California (2005) 158–170
10. Zdancewic, S., Myers, A.C.: Robust declassification. In: CSFW '01: Proceedings of the 14th IEEE Workshop on Computer Security Foundations, IEEE Computer Society (2001) 15–23
11. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: Proceedings of the International Symposium on Software Security (ISSS'03). (2003)
12. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: parameterizing non-interference by abstract interpretation. In: Proceedings of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Venice, Italy (2004) 186–197
13. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. University of California, Berkeley UCB//CSD-05-1396 (Technical report)
14. Volpano, D., Smith, G.: A type-based approach to program security. In Bidoit, M., Dauchet, M., eds.: Theory and Practice of Software Development, 7th International Joint Conference. Volume 1214 of Lecture Notes in Computer Science., Lille, France, Springer-Verlag (1997) 607–621

Author Index

Abramsky, Samson 1
Aiken, Alex 218, 352

Bagnara, Roberto 3, 19
Bruynooghe, Maurice 35

Chen, Guilin 52
Chin, Wei-Ngan 70
Cook, Byron 87

Dimovski, Aleksandar 102

Esparza, Javier 118

Gallagher, John 35
Ganty, Pierre 118
Ghica, Dan R. 102
Glew, Neal 135
Gopan, Denis 186
Gordon, Andrew D. 2
Grothoff, Christian 135

Harren, Matthew 155
Hill, Patricia M. 3
Humbeeck, Wouter Van 35
Hunt, Sebastian 171

Jeannet, Bertrand 186
Jung, Yungbum 203

Kandemir, Mahmut 52
Karakoy, Mustafa 52
Kim, Jaehwang 203
King, Andy 336
Kodumal, John 218

Lazić, Ranko 102

Mastroeni, Isabella 171
Mazzi, Elena 3
Müller-Olm, Markus 235

Narasamdya, Iman 251
Necula, George C. 155
Nguyen, Huu Hai 70

Palsberg, Jens 135
Podelski, Andreas 87, 268

Qin, Shengchao 70

Reps, Thomas 186
Rinard, Martin 70
Rinetzky, Noam 284
Rival, Xavier 303
Rodríguez-Carbonell, Enric 19
Rybalchenko, Andrey 87

Sagiv, Mooly 284
Schwoon, Stefan 118
Secci, Stefano 320
Seidl, Helmut 235
Shin, Jaeho 203
Simon, Axel 336
Spoto, Fausto 320

Terauchi, Tachio 352

Voronkov, Andrei 251

Wies, Thomas 268

Yahav, Eran 284
Yi, Kwangkeun 203

Zaffanella, Enea 3, 19