Lecture Notes in Computer Science 2477

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Manuel V. Hermenegildo Germán Puebla (Eds.)

Static Analysis

9th International Symposium, SAS 2002
Madrid, Spain, September 17-20, 2002
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Manuel V. Hermenegildo
Germán Puebla
Technical University of Madrid, Department of Computer Science
Facultad de Informática, Campus de Montegancedo
28660 Boadilla del Monte, Madrid, Spain
E-mail: {herme, german}@fi.upm.es

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Static analysis : 9th international symposium ; proceedings / SAS 2002, Madrid, Spain, September 17 - 20, 2002. Manuel V. Hermenegildo ; German Puebla (ed.). - Berlin ; Heidelberg ; New York ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2002
(Lecture notes in computer science ; Vol. 2477)
ISBN 3-540-44235-9

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

ISSN 0302-9743
ISBN 3-540-44235-9 Springer-Verlag 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-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York,
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2002
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna e.K.
Printed on acid-free paper SPIN: 10871380 06/3142 5 4 3 2 1 0

Preface

Static analysis is a research area aimed at developing principles and tools for verification and semantics-based manipulation of programs and high-performance implementation of programming languages. The series of Static Analysis Symposia is a forum for the presentation and discussion of advances in the area.

This volume contains the papers accepted for presentation at the Ninth International Static Analysis Symposium (SAS 2002), which was held September 17-20, 2002 in Madrid, Spain. Previous SAS symposia were held in Paris, France (LNCS 2126), Santa Barbara, CA, USA (LNCS 1824), Venice, Italy (LNCS 1694), Pisa, Italy (LNCS 1503), Paris, France (LNCS 1302), Aachen, Germany (LNCS 1145), Glasgow, UK (LNCS 983), Namur, Belgium (LNCS 864), following the international workshop WSA in Padova, Italy (LNCS 724), Bordeaux, France (Bigre Vol. 81-82) and JTASPEFL/WSA, Bordeaux, France (Bigre Vol. 74).

In response to the call for papers, 86 contributions were submitted from 12 different countries. Following on-line discussions, the Program Committee met in Madrid on June 22, 2002, and selected 32 papers, basing this choice on their scientific quality and relevance to the symposium. Each paper was reviewed by at least three Program Committee members or external referees. In addition to the contributed papers, this volume includes abstracts of invited talks by Thomas Reps (University of Wisconsin, USA), Robert Hall (AT&T Labs Research, USA), and Javier Esparza (University of Edinburgh, United Kingdom).

On behalf of the Program Committee, the Program Chairs would like to thank all the authors who submitted papers and all the external referees for their careful work in the reviewing process. The support of the several funding agencies, listed later in the volume, which provided part of the funds for the organization of the conference is also gratefully acknowledged.

On this occasion, SAS was held concurrently with the 2002 International Workshop on Logic-Based Program Development and Transformation and the 2002 APPIA-GULP-PRODE Joint Conference on Declarative Programming as well as several workshops. We would like to express our gratitude to the members of the Local Organizing Committees of all the conferences co-located with SAS 2002 and the members of the organizing departments and institutions involved for the invaluable help and enthusiasm provided throughout the preparation and organization of the joint event.

July 2002

Manuel Hermenegildo
Germán Puebla

Organization

Program Committee

Manuel Hermenegildo (Chair)	Technical University of Madrid, Spain
Germán Puebla (Co-chair)	Technical University of Madrid, Spain
Radhia Cousot	École Polytéchnique, France
Saumya Debray	The University of Arizona, USA
Manuel Fahndrich	Microsoft Research, USA
Roberto Giacobazzi	University of Verona, Italy
Chris Hankin	Imperial College, UK
Giorgio Levi	University of Pisa, Italy
Kim Marriott	Monash University, Australia
Alan Mycroft	Cambridge University, United Kingdom
Ganesan Ramalingam	IBM T.J. Watson Research Center, USA
Martin Rinard	Massachusetts Institute of Technology, USA
Shmuel Sagiv	Tel-Aviv University, Israel
Reinhard Wilhelm	Universität des Saarlandes, Germany

Organizing Committee

SAS 2002 was organized by the following members of the School of Computer Science of the Technical University of Madrid:

Germán Puebla (Local Chair)	Jose Manuel Gómez
Manuel Hermenegildo (Local Co-chair)	Pedro López
Astrid Beascoa	José Morales
Francisco Bueno	Oscar Portela
Jesús Correas	Claudio Vaucheret

Referees

Scott Armenian	Jesús Correas
Roberto Bagnara	Patrick Cousot
Roberto Barbuti	Pierpaolo Degano
Jörg Bauer	Stephan Diehl
Bruno Blanchet	Alessandra Di Pierro
Bard Bloom	Nurit Dor
Chiara Bodei	Robert Ennals
Daniel Cabeza	Jérôme Feret
Manuel Carro	Gian L. Ferrari
Michael Codish	Gilberto Filé

Alessandro Fin
Steve Fink
Riccardo Focardi
Jeffrey S. Foster
Simon Frankau
Maurizio Gabbrielli
John Gallagher
María García de la Banda
Samir Genaim
Deepak Goyal
Roberta Gori
Dave Grove
Orna Grumberg
René R. Hansen
Nevin Heintze
Pat M. Hill
Mike Hind
Yoram Hirshfeld
Charles Hymans
Gerda Janssens
Bertrand Jeannet
Neil Johnson
Andy King
Uday Khedker
Peeter Laud
Michael Leuschel
Francesca Levi
Annie Liu
Francesco Logozzo
Pedro López-García
Andrea Maggiolo-Schettini
Roman Manevich
Julio Mariño
Damien Massé
Isabella Mastroeni
Laurent Mauborgne
Nancy Mazur
Fred Mesnard
Massimo Merro
Antoine Miné
David Monniaux
Susana Muñoz-Hernández
Nicholas Nethercote
Joachim Niehren
Hans de Nivelle
Robert O'Callahan
Andreas Podelski
Christian Probst
Francesco Ranzato
Xavier Rival
David Sands
Peter Schachte
David Scott
Francesca Scozzari
Helmut Seidl
Peter Sewell
Ran Shaham
Richard Sharp
Roberto Segala
Jeremy Singer
Harald Søndergaard
Fausto Spoto
Vugranam. C. Sreedhar
Martin Sulzmann
Peter Stuckey
Henrik Theiling
Stephan Thesing
Eben Upton
Pawel Urzyczyn
Wim Vanhoof
Claudio Vaucheret
Sven Verdoolaege
Herbert Wiklicky
Sebastian Winkel
Jessie Xu
Eran Yahav
Enea Zaffanella
Lenore Zuck

Sponsoring Institutions

The SAS 2002 conference was supported by the following organizations:

- Spanish Ministry of Science and Technology.
- Technical University of Madrid (UPM).
- Department of Computer Science, UPM.
- Department of Industrial Engineering, UPM.
- European Commission – Information Society Technologies Programme.
- CoLogNet, the European Network of Excellence in Computational Logic.

The SAS 2002 conference was organized in cooperation with the European Association for Programming Languages and Systems (EAPLS).

Table of Contents

Static Program Analysis via 3-Valued Logic 1
Thomas Reps

Open Modeling in Multi-stakeholder Distributed Systems: Research and Tool Challenges 2
Robert J. Hall

An Algebraic Approach to the Static Analysis of Concurrent Software 3
Javier Esparza

Polynomial Constants Are Decidable 4
Markus Müller-Olm, Helmut Seidl

Normalizable Horn Clauses, Strongly Recognizable Relations, and Spi 20
Flemming Nielson, Hanne Riis Nielson, Helmut Seidl

Nonuniform Alias Analysis of Recursive Data Structures and Arrays 36
Arnaud Venet

Representing and Approximating Transfer Functions in Abstract Interpretation of Hetereogeneous Datatypes 52
B. Jeannet

Automated Verification of Concurrent Linked Lists with Counters 69
Tuba Yavuz-Kahveci, Tevfik Bultan

Backward Type Inference Generalises Type Checking 85
Lunjin Lu, Andy King

More Precise Yet Efficient Type Inference for Logic Programs 102
Claudio Vaucheret, Francisco Bueno

A Few Graph-Based Relational Numerical Abstract Domains 117
Antoine Miné

Static Analysis of the Numerical Stability of Loops 133
Matthieu Martel

On Termination of Logic Programs with Floating Point Computations 151
Alexander Serebrenik, Danny De Schreye

Modular Control Flow Analysis for Libraries 165
Christian W. Probst

An Efficient Inclusion-Based Points-To Analysis for Strictly-Typed Languages 180
John Whaley, Monica S. Lam

Compactly Representing First-Order Structures for Static Analysis 196
R. Manevich, G. Ramalingam, J. Field, D. Goyal, M. Sagiv

Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library 213
Roberto Bagnara, Elisa Ricci, Enea Zaffanella, Patricia M. Hill

Speeding Up Dataflow Analysis Using Flow-Insensitive Pointer Analysis 230
Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, Westley Weimer

Improving the Precision of Equality-Based Dataflow Analyses 247
Erik Ruf

Guaranteed Optimization: Proving Nullspace Properties of Compilers 263
Todd L. Veldhuizen, Andrew Lumsdaine

Representation Analysis for Coercion Placement 278
Karl-Filip Faxén

Pipeline Modeling for Timing Analysis 294
Marc Langenbach, Stephan Thesing, Reinhold Heckmann

Analysing Approximate Confinement under Uniform Attacks 310
Alessandra Di Pierro, Chris Hankin, Herbert Wiklicky

An Improved Constraint-Based System for the Verification of Security Protocols 326
Ricardo Corin, Sandro Etalle

From Secrecy to Authenticity in Security Protocols 342
Bruno Blanchet

Security Typings by Abstract Interpretation 360
Mirko Zanotti

Static Confidentiality Enforcement for Distributed Programs 376
Andrei Sabelfeld, Heiko Mantel

Refinement of LTL Formulas for Abstract Model Checking 395
María del Mar Gallardo, Pedro Merino, Ernesto Pimentel

Making Abstract Model Checking Strongly Preserving 411
Francesco Ranzato, Francesco Tapparo

Semantics for Abstract Interpretation-Based Static Analyzes of Temporal Properties 428
Damien Massé

Checking Safety Properties of Behavioral VHDL Descriptions by Abstract Interpretation 444
Charles Hymans

States vs. Traces in Model Checking by Abstract Interpretation 461
Roberto Giacobazzi, Francesco Ranzato

Reuse of Results in Termination Analysis of Typed Logic Programs 477
Maurice Bruynooghe, Michael Codish, Samir Genaim, Wim Vanhoof

Finiteness Analysis in Polynomial Time 493
Chin Soon Lee

Detecting Optimal Termination Conditions of Logic Programs 509
Fred Mesnard, Etienne Payet, Ulrich Neumerkel

Author Index 527

Static Program Analysis via 3-Valued Logic

Thomas Reps

Computer Sciences Department
University of Wisconsin
reps@cs.wisc.edu

Abstract interpretation serves as a powerful theoretical tool for developing and justifying program-analysis algorithms. It provides a way to establish that information extracted from a program by a program-analysis algorithm is a meaningful characterization of what can occur when the program is executed. Typically, however, it is not an easy task to obtain the appropriate abstract state-transformation functions and to show that they are correct. On the contrary, papers on program analysis often contain exhaustive (and exhausting) proofs to demonstrate that a given abstract semantics provides answers that are safe with respect to a given concrete semantics.

In contrast, the parametric framework for program analysis proposed by Sagiv, Reps, and Wilhelm [POPL '99, TOPLAS (in press)] holds out the promise of allowing different program-analysis tools to be generated automatically from high-level descriptions of what an analysis designer desires. Analyses generated in this fashion would be correct by construction. This framework has two parts:

- A language (3-valued logic) for specifying various properties that an entity may or may not possess, and how these properties are affected by the execution of the different kinds of statements in the programming language.
- A method for generating a static-analysis algorithm from such a description.

The talk will review the principles behind the paradigm of "abstract interpretation via 3-valued logic," discuss recent work to extend the approach, and summarize on-going research aimed at overcoming remaining limitations on the ability to create program-analysis algorithms fully automatically.

(Joint work with M. Sagiv, R. Wilhelm, F. DiMaio, N. Dor, T. Lev-Ami, A. Loginov, A. Mulhern, N. Rinetzky, and E. Yahav.)

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, p. 1, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Open Modeling in Multi-stakeholder Distributed Systems: Research and Tool Challenges

Robert J. Hall

AT&T Labs Research
Florham Park, NJ 07932

The design and use of multi-stakeholder distributed systems (MSDSs – in which nodes are designed or operated by distinct stakeholders having limited knowledge and possibly conflicting goals) challenge our traditional conceptions of requirements engineering and system validation in two ways. First, global inconsistency among high-level requirements forces us to focus on the personalized and time sensitive requirements of a single stakeholder, rather than the more traditional global, time-invariant requirements. Second, any given stakeholder will typically lack detailed knowledge of the behaviors of nodes not under his control; such knowledge is necessary to validating requirements conformance. OpenModel [1] is a research effort whose aim is to address this "ignorance problem" through open, standardized behavioral modeling. In OpenModel, each node of an MSDS such as the Internet email system or a network of web services provides via http (or through a central registry) a behavioral model expressed in terms of shared domain-specific function/object theories. By assembling these models appropriately, tools based upon techniques like scenario simulation, animation, coverage measurement, theorem proving, and model checking can support a wide range of automated software engineering activities: personalized requirements engineering, validation, on-line help, node design, configuration, and evolution.

The OpenModel proposal presents significant research challenges for the fields of requirements engineering, formal methods, ontology engineering, and automated software engineering. This talk will first motivate and briefly describe OpenModel and its context, and then will discuss these challenges as well as some potentially promising future research directions.

References

1. R.J. Hall, Open Modeling in Multi-stakeholder Distributed Systems: Model-based Requirements Engineering for the 21st Century, in *2002 Workshop on the State of the Art in Automated Software Engineering*, U.C.Irvine, Institute for Software Research `http://www.isr.uci.edu/events/ASE-Workshop-2002/`. Paper also available at `ftp://ftp.research.att.com/dist/hall/papers/openmodel/openmodel-asewshp02.pdf`

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, p. 2, 2002.
© Springer-Verlag Berlin Heidelberg 2002

An Algebraic Approach to the Static Analysis of Concurrent Software

Javier Esparza

Laboratory for Foundations of Computer Science
University of Edinburgh
www.dcs.ed.ac.uk/home/jav

In the last years a group of colleagues and I have developed a new approach to interprocedural program analysis based on ideas born in the model-checking community [2,3,4,6]. The Moped tool [8] is an implementation of most of this work. In interaction with the SLAM toolkit [1], Moped has been applied to the analysis of programs with thousands of lines of C code.

In [5] we have succeeded in extending the approach to the interprocedural analysis of parallel programs in which tasks do not synchronise, or synchronise only at termination. In this talk I present our current efforts to extend the approach so that it can also encompass synchronisation. I intend to show that some beautiful results of the theory of commutative semirings and Kleene algebras [4] could well be the formal basis of flexible software able to perform a large variety of analysis of different precision and cost.

References

1. T. Ball and S. Rajamani: The SLAM Project: Debugging System Software via Static Analysis Conference Record of POPL '02, 1–3, ACM Press (2002).
2. A. Bouajjani, J. Esparza and O. Maler: Reachability Analysis of Pushdown Automata: Application to Model-Checking. Proceedings of CONCUR '97, LNCS 1243, 135–150 (1997).
3. J. Esparza, D. Hansel, P. Rossmanith and S. Schwoon: Efficient Model Checking Algorithms for Pushdown Systems. Proceedings of CAV '00, LNCS 1855, 232–247 (2000).
4. J. Esparza and J. Knoop: An Automata-theoretic Approach to Interprocedural Dataflow Analysis. Proceedings of FOSSACS '99, LNCS 1578, 14–30 (1999).
5. J.Esparza and A. Podelski: Efficient algorithms for pre^* and $post^*$ on Interprocedural Parallel Flow Graphs. Conference Record of POPL '00, 1–11, ACM Press (2000).
6. J. Esparza and S. Schwoon: A BBD-Based Model Checker for Recursive Programs. Proceedings of CAV '01, LNCS 2102, 324–336 (2001).
7. M.W. Hopkins and D.C. Kozen: Parikh's Theorem in Commutative Kleene Algebra Proceedings of LICS'99, IEEE, 394–401 (1999)
8. Moped - A Model-Checker for Pushdown Systems. Writen and maintained by Stefan Schwoon. Code available at http://www7.in.tum.de/~schwoon/moped.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, p. 3, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Polynomial Constants Are Decidable*

Markus Müller-Olm[1] and Helmut Seidl[2]

[1] University of Dortmund, FB 4, LS5, 44221 Dortmund, Germany
mmo@ls5.cs.uni-dortmund.de
[2] Trier University, FB 4-Informatik, 54286 Trier, Germany
seidl@uni-trier.de

Abstract. Constant propagation aims at identifying expressions that always yield a unique constant value at run-time. It is well-known that constant propagation is undecidable for programs working on integers even if guards are ignored as in non-deterministic flow graphs. We show that *polynomial constants* are decidable in non-deterministic flow graphs. In polynomial constant propagation, assignment statements that use the operators $+, -, *$ are interpreted exactly but all assignments that use other operators are conservatively interpreted as non-deterministic assignments.
We present a generic algorithm for constant propagation via a symbolic weakest precondition computation and show how this generic algorithm can be instantiated for polynomial constant propagation by exploiting techniques from computable ring theory.

1 Introduction

Constant propagation is one of the most widely used optimizations in practical optimizing compilers (cf. [1,9,15]). Its goal is to replace expressions that always yield a unique constant value at run-time by this value. This both speeds up execution and reduces code size. Even more importantly, it can enable powerful further transformations like elimination of dynamically unreachable branches.

In order to come to grips with fundamental computability problems one often abstracts guarded branching to non-deterministic branching in program analysis. But even this abstraction leaves constant propagation undecidable for programs working on integer variables. This has already been observed in the seventies independently by Hecht [9] and by Reif and Lewis [17]. We briefly recall the construction of Reif and Lewis. It is based on a reduction of Hilbert's tenth problem, whether a (multivariate) polynomial has a zero in the natural numbers, a very famous undecidable problem [12].

Assume given a (non-zero) polynomial $p(x_1, \ldots, x_n)$ in n variables $x_1, \ldots, x_n$ with integer coefficients and consider the (non-deterministic) program in Fig. 1. The initializations and the loops choose arbitrary natural values for the variables x_i. If the chosen values constitute a zero of $p(x_1, \ldots, x_n)$, then $p(x_1, \ldots, x_n)^2 +$

* The work was supported by the RTD project IST-1999-20527 "DAEDALUS" of the European FP5 programme.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 4–19, 2002.
© Springer-Verlag Berlin Heidelberg 2002

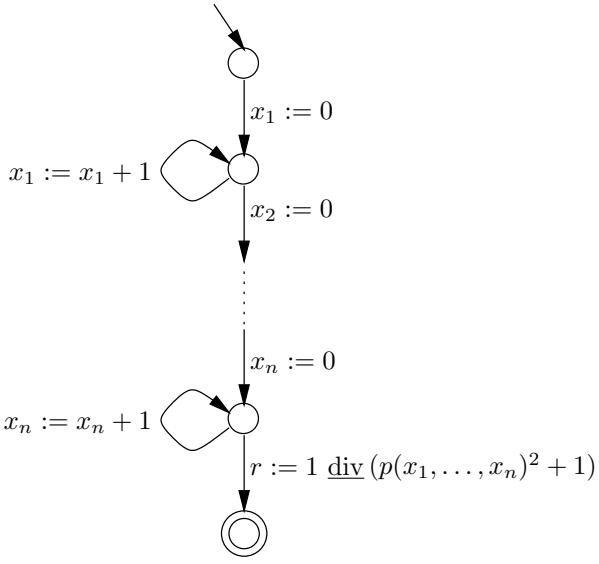


Fig. 1. Undecidability of constant detection; the reduction of Reif and Lewis [17].

$1 = 1$ and r is set to 1. Otherwise, $p(x_1, \ldots, x_n)^2 + 1 \geq 2$ such that r is set to 0. Therefore, r is a constant (of value 0) at the end of the program if and only if $p(x_1, \ldots, x_n)$ does not have a natural zero.

On the other hand there are well-known and well-defined classes of constants that can be detected, even efficiently. In *copy constant detection* [7] only assignments of the form $x := c$, where c is either an (integer) constant or a program variable are interpreted; assignments with composite expressions on the right hand side are conservatively assumed to make a variable non-constant. In *linear constants* [18] also assignments of the form $x := a \cdot y + b$, where a and b are integer constants and y is a program variable, are interpreted. Another decidable class of constants are *finite constants* [19]. This motivated Müller-Olm and Rüthing [16] to study the complexity of constant propagation for classes that derive from interpreting a subset of integer operators.

An interesting question they left open concerns the class of constants obtained by interpreting just $+, -, *$, i.e., all standard integer operators except of division operators. While they called the corresponding class of constants $+, -, *$-constants, we prefer the term *polynomial constants*, because with these operators we can just write (multivariate) polynomials. The detection problem for polynomial constants is PSPACE-hard [16] but no upper bound is known. In the current paper we show that polynomial constants are decidable. In order to obtain this result we apply results from computable ideal theory. This decidability result suggests that the division operator is the real cause for undecidability of general constant detection.

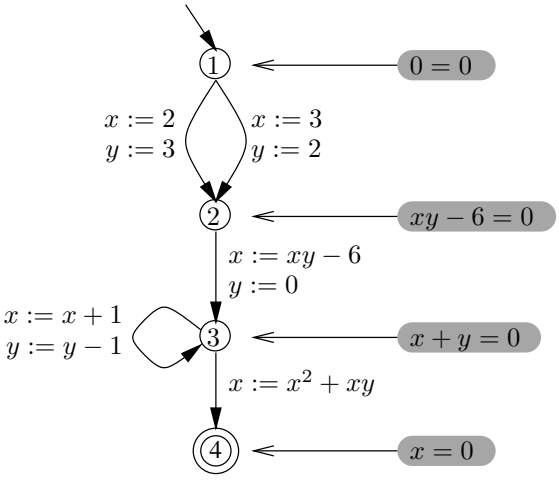


Fig. 2. An example flow graph.

The paper is organized as follows. In the next section we illustrate our algorithm at an example that cannot be handled by other constant propagation algorithms. Afterwards we define flow graphs, the constant propagation problem, and weakened versions thereof. This enables us to define polynomial constants formally. In Section 4 we present a generic algorithmic idea for constant propagation via a symbolic weakest precondition computation and discuss the requirements for making it effective. In Section 5 we recall results from computable ring theory that are needed in our algorithm and proceed with further observations on $\mathbb{Z}[x_1, \ldots, x_n]$, the set of multivariate polynomials with integer coefficients. These results are exploited in Section 7, where we show how the generic algorithm from Section 4 can be instantiated for polynomial constant propagation. We finish the paper with concluding remarks and directions for future research.

2 An Example

Let us illustrate the power of our algorithm. In the flow graph in Fig. 2, x is a constant (of value 0) at node 4, but no known constant propagation algorithms can handle this example. Standard *simple constant propagation* [1] propagates variable assignments through the program, in which each variable is assigned either a definite value or a special value unknown. Simple constant propagation computes the value unknown at program point 2 for both x and y and cannot recover from this loss of precision. More powerful constant propagation algorithms like the algorithm for *linear constants* [18] or Presburger constants [16] cannot handle the expressions $xy - 6$ and $x^2 + xy$. Even the EXPTIME algorithm of Knoop and Steffen for finite constants [19] cannot handle this example because

no finite unfolding of the loop suffices to prove constancy of $x^2 + xy$ after the loop.

Before we turn to the technical development, we discuss informally how our algorithm detects constancy of x at node 4. In a first phase some path from node 1 to 4 is executed, e.g., $\langle 1, 2, 3, 4\rangle$, and the value of x, viz. 0, after execution of this path is computed. This implies that x can only be a constant of value 0 at program point 4 – if it is a constant at all. In order to check this, our algorithm propagates the assertion $\mathcal{A}_0 : x = 0$ backwards from node 4 towards the start node which amounts to a symbolic computation of the weakest precondition of $\mathcal{A}_0$ at node 4. Propagation over statement $x := x^2 + xy$ results in the assertion $\mathcal{A}_1 : x^2 + xy = 0$. Assertion $\mathcal{A}_1$ is then propagated through the loop. This results in the assertion $\mathcal{A}_2 : (x + 1)^2 + (x + 1)(y - 1) = 0$ that can be simplified to $\mathcal{A}_3 : x^2 + xy + x + y = 0$. Both $\mathcal{A}_1$ and $\mathcal{A}_3$ must be valid at program point 3 in order to guarantee validity of $\mathcal{A}_0$ at program point 4. We can simplify the assertion $\mathcal{A}_1 \wedge \mathcal{A}_3$: because $\mathcal{A}_1$ guarantees that $x^2 + xy$ equals 0, we can simplifiy $\mathcal{A}_3$ to $\mathcal{A}_4 : x + y = 0$; now, as $\mathcal{A}_1$ can be written in the form $x(x + y) = 0$, we see that $\mathcal{A}_1$ is indeed implied by $\mathcal{A}_4$. Thus, validity of $\mathcal{A}_4$ suffices to guarantee both $\mathcal{A}_1$ and $\mathcal{A}_3$. $\mathcal{A}_4$ is again propagated through the loop; this results in $\mathcal{A}_4$ again; hence no further propagation through the loop is necessary. In this way propagation goes on and results in the assertions shown in Fig. 2. The assertion computed for the start node, $0 = 0$ is universally valid; this proves that x is indeed a constant of value 0 at node 4.

In the algorithm developed in the remainder of this paper, assertions are represented by Gröbner bases of ideals in the polynomial ring $\mathbb{Z}[x_1, \ldots, x_n]$. As Gröbner bases are a *canonic* representation this also takes care of simplifications.

3 The Framework

Flow Graphs. Suppose given a finite set of variables $X = \{x_1, \ldots, x_n\}$. Let Expr be a set of expressions over X; the precise nature of expressions is immaterial at the moment. A (deterministic) assignment is a pair consisting of a variable and an expression written as $x := t$; the set of assignment statements is denoted by Asg. A non-deterministic assignment statement consists of a variable and is written $x :=?$; the set of nondeterministic assignment statements is denoted by NAsg.

A (non-deterministic) flow graph is a structure $G = (N, E, A, \mathbf{s}, \mathbf{e})$ with finite node set N, edge set $E \subseteq N \times N$, a unique start node $\mathbf{s} \in N$, and a unique end node $\mathbf{e} \in N$. We assume that each program point $u \in N$ lies on a path from $\mathbf{s}$ to $\mathbf{e}$. The mapping $A : E \to \mathsf{Asg} \cup \mathsf{NAsg} \cup \{\mathsf{skip}\}$ associates each edge with a deterministic or non-deterministic assignment statement or the statement skip. Edges represent the branching structure and the statements of a program, while nodes represent program points. The set of successors of program point $u \in N$ is $Succ[u] = \{v \mid (u, v) \in E\}$.

A *path* reaching a given program point $u \in N$ is a sequence of edges $p = \langle e_1, \ldots, e_k\rangle$ with $e_i = (u_i, v_i) \in E$ such that $u_1 = \mathbf{s}$, $v_k = u$, and $v_i = u_{i+1}$ for

$1 \leq i < k$. In addition $p = \varepsilon$, the empty sequence, is a path reaching the start node $\mathbf{s}$. We write $\mathsf{R}[u]$ for the set of paths reaching u.

Let Val be a set of values. A mapping $\sigma : X \to \mathsf{Val}$ that assigns a value to each variable is called a *state*; we write $\Sigma = \{\sigma \mid \sigma : X \to \mathsf{Val}\}$ for the set of states. For $x \in X$, $d \in \mathsf{Val}$, and $\sigma \in \Sigma$ we write $\sigma[x \mapsto d]$ for the state that maps x to d and coincides for the other variables with σ. We assume a fixed interpretation for the operators used in terms and we assume that the value of term t in state σ, which we denote by t^σ, is defined in the standard way.

In order to accommodate non-deterministic assignments we interpret statements by relations on Σ rather than functions. The relation associated with assignment statement $x := t$ is $[\![x := t]\!] \stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid \sigma' = \sigma[x \mapsto t^\sigma]\}$; the relation associated with non-deterministic assignment $x :=?$ is $[\![x :=?]\!] \stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid \exists d \in \mathsf{Val} : \sigma' = \sigma[x \mapsto d]\}$; and the relation associated with skip is the identity: $[\![\mathsf{skip}]\!] \stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid \sigma = \sigma'\}$. This local interpretation of statements is straightforwardly extended to paths $p = \langle e_1, \ldots, e_k \rangle \in E^*$ as follows: $[\![p]\!] = [\![A(e_1)]\!] ; \ldots ; [\![A(e_k)]\!]$, where ; denotes relational composition.

Constant Propagation. A variable $x \in X$ is a *constant* at program point $u \in N$ if there is $d \in \mathsf{Val}$ such that $\sigma(x) = d$ for all $p \in \mathsf{R}[u]$, $(\sigma_0, \sigma) \in [\![p]\!]$. Arbitrary choice of initial state σ_0 reflects that we do not know the state in which the program is started.

We can weaken the demands for a constant detection algorithm by selecting a certain subset of expressions $S \subseteq \mathsf{Expr}$ that are interpreted precisely and assuming conservatively that assignments whose right hand does not belong to S assign an arbitrary value to their respective target variable. This idea can be made formal as follows.

For a given flow graph $G = (N, E, A, \mathbf{s}, \mathbf{e})$ and subset of expressions $S \subseteq \mathsf{Expr}$, let $G_S = (N, E, A_S, \mathbf{s}, \mathbf{e})$ be the flow graph with the same underlying graph but with the following weakened edge annotation:

$$A_S(u, v) = \begin{cases} x :=?\,, & \text{if } A(u, v) = (x := t) \text{ and } t \notin S \\ A(u, v)\,, & \text{otherwise} \end{cases}$$

A variable $x \in X$ is then called an *S-constant* at program point $u \in N$ in flow graph G if it is a constant at u in the weakened flow graph G_S. Clearly, if x is an S-constant at u it is also a constant at u but not vice versa. The *detection problem for S-constants* is the problem of deciding for a given set of variables X, flow graph G, variable x, and program point u whether x is an S-constant at u or not.

To study weakened versions of constant detection problems is particularly interesting for programs computing on the integers, i.e., if Expr is the set of integer expressions formed from integer constants and variables with the standard operators $+, -, *, \underline{\text{div}}, \underline{\text{mod}}$: we have seen in the introduction that the general constant detection problem is undecidable in this case.

Let us discuss some examples for illustration. S-constants with respect to the set $S = X \cup \mathbb{Z}$, i.e., the set of non-composite expressions, are known as *copy*

constants [7]. S-constants with respect to the set $S = \{a*x+b \mid a, b \in \mathbb{Z}, x \in X\}$ are known as *linear constants* [18]. In this paper we tackle constants with respect to the set $S = \mathbb{Z}[x_1, \ldots, x_n]$, the set of multivariate polynomials in the variables $x_1, \ldots, x_n$ with coefficients in $\mathbb{Z}$, which we call *polynomial constants*.

We should emphasize two points about the above framework that make the construction of S-constant detection algorithms more challenging. Firstly, in contrast to the setup in [16], we allow assignment statements, whose right hand side does not belong to S. They are interpreted as non-deterministic assignments. Forbidding them is adequate for studying lower complexity bounds for analysis questions, which is the main concern of [16]. It is less adequate when we are concerned with detection algorithms because in practice we want to detect S-constants in the context of other code.

Secondly, a variable can be an S-constant although its value statically depends on an expression that is not in S. As a simple example consider the flow graph in Fig. 3 and assume that the expressions 0 and $y - y$ belong to S but e does not. Because $y - y$ is 0 for any value $y \in \mathbb{Z}$, an S-constant detection algorithm must identify x as a constant (of value 0), although its value statically depends on the uninterpreted expression e. Hence, S-constant detection must handle arithmetic properties of the expressions in S.

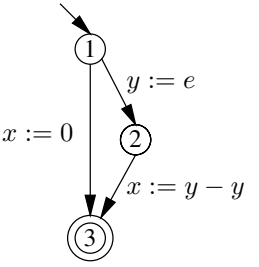


Fig. 3. An S-constant.

4 A Generic Algorithm

Suppose given a variable $x \in X$ and a program point $w \in N$. In this chapter we describe a generic algorithm for deciding whether x is an S-constant at w or not. While standard constant propagation works by forward propagation of variable assignments, we use a three phase algorithm that employs a backwards propagation of assertions. For the moment we can think of assertions as predicates on states as in program verification.

Phase 1: In the first phase we follow an arbitrary cycle-free path from **s** to w, for instance using depth-first search, and compute the value c, referred to as the *candidate value*, that x holds after this path is executed. This implies that, if x is a constant at w, it must be a constant of value c.

Phase 2: In the second phase we compute the weakest precondition for the assertion $x = c$ at program point w in G_S by means of a backwards data flow analysis.

Phase 3: Finally, we check whether the computed weakest precondition for $x = c$ at w is `true`, i.e., is valid for all states.

It is obvious that this algorithm is correct. The problem is that Phase 2 and 3 are in general not effective. However, as only assignments of a restricted form appear in G_S, the algorithm becomes effective for certain sets S, if assertions

are represented appropriately. In the remainder of this section we analyze the requirements for adequate representations. For this purpose, we first characterize weakest preconditions in flow graphs.

Semantically, an *assertion* is a subset of states $\phi \subseteq \Sigma$. Given an assertion ϕ and a statement s, the *weakest precondition* of s for ϕ, $\mathsf{wp}(s)(\phi)$, is the largest assertion ϕ' such that execution of s from all states in ϕ' is guaranteed to terminate only in states in ϕ.[1] The following identities for the weakest precondition of assignment and skip statements are well-known:

$$\begin{aligned}
\mathsf{wp}(x := e)(\phi) &\stackrel{\text{def}}{=} \phi[e/x] \stackrel{\text{def}}{=} \{\sigma \mid \sigma[x \mapsto e^\sigma] \in \phi\} \\
\mathsf{wp}(x :=?)(\phi) &\stackrel{\text{def}}{=} \forall x(\phi) \stackrel{\text{def}}{=} \{\sigma \mid \forall d \in \mathbb{Z} : \sigma[x \mapsto d] \in \phi\} \\
\mathsf{wp}(\mathsf{skip})(\phi) &\stackrel{\text{def}}{=} \phi
\end{aligned}$$

These identities characterize weakest preconditions of basic statements. Let us now consider the following more general situation in a given flow graph $G = (N, E, A, \mathbf{s}, \mathbf{e})$: we are given an assertion $\phi \subseteq \Sigma$ as well as a program point $w \in N$ and we are interested in the weakest precondition that guarantees validity of ϕ whenever execution reaches w. The latter can be characterized as follows.

Let $W_0[w] = \phi$ and $W_0[u] = \Sigma$ and consider the following equation system consisting of one equation for each program point $u \in N$:

$$\mathbf{W}[u] \;=\; W_0[u] \cap \bigcap_{v \in Succ[u]} \mathsf{wp}(A(u,v))(\mathbf{W}[v])\,. \tag{1}$$

By the Knaster-Tarski fixpoint theorem, this equation system has a largest solution (w.r.t. subset inclusion) because $\mathsf{wp}(s)$ is well-known to be monotonic. By abuse of notation, we denote the weakest solution by the same letter $\mathbf{W}[u]$. For each program point $u \in N$, $\mathbf{W}[u]$ is the weakest assertion such that execution starting from u with any state in $\mathbf{W}[u]$ guarantees that ϕ holds whenever execution reaches w. In particular, $\mathbf{W}[\mathbf{s}]$ is the weakest precondition for validity of ϕ at w. The intuition underlying equation (1) is the following: firstly, $W_0[u]$ must be implied by $\mathbf{W}[u]$ and, secondly, for all successors v, we must guarantee that their associated condition $\mathbf{W}[u]$ is valid after execution of the statement $A(u,v)$ associated with the edge (u,v); hence $\mathsf{wp}(A(u,v))(\mathbf{W}[v])$ must be valid at u too.

For two reasons, the above equation system cannot be solved directly in general: firstly, because assertions may be infinite sets of states they cannot be represented explicitly; secondly, there are infinitely long descending chains of assertions such that standard fixpoint iteration does not terminate in general.

In order to construct an algorithm that detects S-constants we represent assertions by the members of a lattice $(\mathbb{D}, \sqsubseteq)$. Let us assume that $\gamma : \mathbb{D} \to 2^\Sigma$ captures how the lattice element represent assertions. First of all, we require

[1] In the sense of Dijkstra [6] this is the weakest *liberal* precondition as it does not guarantee termination. For simplicity we omit the qualifying prefix "liberal" in this paper.

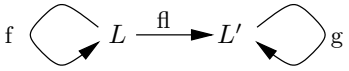


Fig. 4. Situation in the transfer lemma.

(a) $\mathbb{D}$ has no infinite decreasing chains, i.e., there is no infinite chain $d_1 \sqsupseteq d_2 \sqsupseteq d_3 \sqsupseteq \ldots$.

This guarantees that maximal fixpoints of monotonic functions can effectively be computed by standard fixpoint iteration. Secondly, we suppose

(b) γ is universally conjunctive, i.e., $\gamma(\sqcap X) = \bigcap\{\gamma(d) \mid d \in X\}$ for all $X \subseteq \mathbb{D}$.

The most important reason for making this assumption is that it ensures that we can validly compute on representations without loosing precision: if we precisely mirror the equations characterizing weakest preconditions on representations, the largest solution of the resulting equation system on representations characterizes the representation of the weakest precondition by the following well-known lemma. It appears in the literature (for the dual situation of least fixpoints) under the name *Transfer Lemma* [2] or *μ-Fusion Rule* [11].

Lemma 1. *Suppose L, L' are complete lattices, $f : L \to L$ and $g : L' \to L'$ are monotonic functions and $\gamma : L \to L'$ (cf. Fig. 4).*

If γ is universally conjunctive and $\gamma \circ f = g \circ \gamma$ then $\gamma(\nu f) = \nu g$, where νf and νg are the largest fixpoints of f and g, respectively.

We must mirror the elements comprising the equation system characterizing weakest preconditions on representations precisely. Firstly, we must represent the start value, W_0. Universal conjunctivity of γ implies that $\gamma(\top) = \Sigma$, i.e., the top value of $\mathbb{D}$ is a precise representation of Σ. In addition, we require:

(c) Assertion $x = c$ can be represented precisely: for each $x \in X$, $c \in \mathsf{Val}$ we can effectively determine $d_{x=c} \in \mathbb{D}$ with $\gamma(d_{x=c}) = \{\sigma \in \Sigma \mid \sigma(x) = c\}$.

Secondly, we need effective representations for the operators appearing in equations. Requirement (b) implies that the meet operation of $\mathbb{D}$ precisely abstracts intersection of assertions. In order to enable effective computation of intersections, we require in addition:

(d) for given $d, d' \in \mathbb{D}$, we can effectively compute $d \sqcap d'$.

By induction this implies that we can compute finite meets $d_1 \sqcap \ldots \sqcap d_k$ effectively.

The only remaining operations on assertions are the weakest precondition transformers of basic statements. We must represent $\mathsf{wp}(x := t)$ for expressions $t \in S$, which is the substitution operator $(\cdot)[t/x]$ on assertions. As the S-constant detection algorithm computes the weakest precondition in weakened flow graph G_S, assignments $x := t$ with $t \notin S$ do not occur.

(e) There is a computable substitution operation $(\cdot)[t/x] : \mathbb{D} \to \mathbb{D}$ for each $x \in X$, $t \in S$, which satisfies $\gamma(d[t/x]) = \gamma(d)[t/x]$ for all $d \in \mathbb{D}$.

Obviously, $\mathsf{wp}(\mathsf{skip})$, the identity, is precisely represented by the identity on R. Thus, it remains to represent $\mathsf{wp}(x :=?)$:

(f) There is a computable projection operation $proj_i : \mathbb{D} \to \mathbb{D}$ for each variable $x_i \in X$ such that $\gamma(proj_i(d)) = \forall x_i(\gamma(d))$ for all $d \in \mathbb{D}$.

Finally, we need the following in order to make Phase 3 of the algorithm effective.

(g) Assertion true is decidable, i.e., there is a decision procedure that decides for a given $d \in \mathbb{D}$, whether $\gamma(d) = \Sigma$ or not.

If, for a given set $S \subseteq \mathsf{Expr}$, we can find a lattice satisfying requirements (a)–(g), we can effectively execute the three phase algorithm from the beginning of this section by representing assertions by elements from this lattice. This results in a detection algorithm for S-constants.

In this paper we are interested in detection of polynomial constants. Thus, from now on, let $\mathsf{Val} = \mathbb{Z}$ and $S = \mathbb{Z}[x_1, \ldots, x_n]$. The key idea for the detection of polynomial constants is to represent assertions by the zeros of ideals in the polynomial ring $\mathbb{Z}[x_1, \ldots, x_n]$ and to apply techniques from computable ring theory. A full introduction to this area is beyond the scope of this paper but we recall the facts needed in the next section and make some additional observations in Section 6. Accessible introductions can be found in standard textbooks on computer algebra. The case of polynomial rings over fields is covered, e.g., by [5,8,20], while [14] treats the more general case of polynomial rings over rings, that is of relevance here, as $\mathbb{Z}$ is an integral domain but not a field.

5 A Primer on Computable Ideal Theory

Recall that $\mathbb{Z}$ together with addition and multiplication forms a commutative ring, i.e., a structure $(R, +, \cdot)$ with a non-empty set R and two inner operations $+$ and $\cdot$ such that $(R, +)$ is an Abelian group, $\cdot$ is associative and commutative, and the distributive law $a \cdot (b + c) = a \cdot b + a \cdot c$ is valid for all $a, b, c \in R$. On the set of polynomials, $\mathbb{Z}[x_1, \ldots, x_n]$, we can define addition and multiplication operations in the standard way; this makes $\mathbb{Z}[x_1, \ldots, x_n]$ a commutative ring as well.

A non-empty subset $I \subseteq R$ of a ring R is called an *ideal* if $a + b \in I$ and $r \cdot a \in I$ for all $a, b \in I$, $r \in R$. The ideal *generated* by a subset $B \subseteq R$ is

$$(B) = \{r_1 \cdot b_1 + \ldots + r_k \cdot b_k \mid r_1, \ldots, r_k \in R, b_1, \ldots, b_k \in B\},$$

and B is called a *basis* or *generating system* of I if $I = (B)$. An ideal is called *finitely generated* if it has a finite basis $B = \{b_1, \ldots, b_m\}$. Hilbert's famous basis theorem tells us that $\mathbb{Z}[x_1, \ldots, x_n]$ is *Noetherian*, since $\mathbb{Z}$ is Noetherian, i.e., that there are no infinitely long strictly increasing chains $I_1 \subset I_2 \subset I_3 \subset \ldots$ of

ideals in $\mathbb{Z}[x_1, \ldots, x_n]$. This implies that every ideal of $\mathbb{Z}[x_1, \ldots, x_n]$ is finitely generated.

It is crucial for our algorithm that we can compute effectively with ideals. While Hilbert's basis theorem ensures that we can represent every ideal of $\mathbb{Z}[x_1, \ldots, x_n]$ by a finite basis, in itself it does not give effective procedures for basic questions like membership tests or equality tests of ideals represented in this way. Indeed, Hilbert's proof of the basis theorem was famous (and controversial) at its time for its non-constructive nature.

Fortunately, the theory of Gröbner bases and the Buchberger algorithm provides a solution for some of these problems. While a complete presentation of this theory is way beyond the scope of this paper – the interested reader is pointed to the books mentioned above – a few sentences are in order here. A Gröbner basis is a basis for an ideal that has particularly nice properties. It can effectively be computed from any given finite basis by the Buchberger algorithm. There is a natural notion of reduction of a polynomial with respect to a set of polynomials such that reduction of a polynomial p with respect to a Gröbner basis always terminates and yields a unique result. This result is the zero polynomial if and only if p belongs to the ideal represented by the Gröbner basis. Hence reduction with respect to a Gröbner basis yields an effective membership test, that in turn can be used to check equality and inclusion of ideals.

In the terminology of [14], $\mathbb{Z}[x_1, \ldots, x_n]$ is a *strongly computable ring.* This implies that the following operations are computable for polynomials $p \in \mathbb{Z}[x_1, \ldots, x_n]$ and ideals $I, I' \subseteq \mathbb{Z}[x_1, \ldots, x_n]$ given by finite bases B, B', cf. [14]:

Ideal membership: Given an ideal I and a polynomial p. Is $p \in I$?
Ideal inclusion: Given two ideals I, I'. Is $I \subseteq I'$?
Ideal equality: Given two ideals I, I'. Is $I = I'$?
Sum of ideals: Given two ideals I, I'. Compute a basis for $I + I' \stackrel{\text{def}}{=} \{p + p' \mid p \in I, p' \in I'\}$. As a matter of fact, $I + I' = (B \cup B')$.
Intersection of ideals: Given two ideals I, I'. Compute a basis for $I \cap I'$.

It is well-known that $I + I'$ and $I \cap I'$ are again ideals if I and I' are. We can use the above operations as basic operations in our algorithms.

6 More on $\mathbb{Z}[x_1, \ldots, x_n]$

$\mathbb{Z}[x_1, \ldots, x_n]$ as Complete Lattice. Interestingly, the ideals in $\mathbb{Z}[x_1, \ldots, x_n]$ form also a complete lattice under subset inclusion $\subseteq$. Suppose given a set $\mathcal{I}$ of ideals in $\mathbb{Z}[x_1, \ldots, x_n]$. Then the largest ideal contained in all ideals in $\mathcal{I}$ obviously is $\bigcap \mathcal{I}$, and the smallest ideal that contains all ideals in $\mathcal{I}$ is $\sum \mathcal{I} := \{r_1 \cdot a_1 + \ldots + r_k \cdot a_k \mid r_1, \ldots, r_k \in \mathbb{Z}[x_1, \ldots, x_n], a_1, \ldots, a_k \in \bigcup \mathcal{I}\}$. The least element of the lattice is the zero ideal $\{0\}$ that consists only of the zero polynomial and the largest element is $\mathbb{Z}[x_1, \ldots, x_n]$. While this lattice does not have finite height it is Noetherian by Hilbert's basis theorem such that we can effectively compute least fixpoints of monotonic functions on ideals of $\mathbb{Z}[x_1, \ldots, x_n]$ by standard fixpoint iteration.

Zeros. We represent assertions by the zeros of ideals in our algorithm. A state σ is called a *zero* of polynomial p if $p^\sigma = 0$; we denote the set of zeros of polynomial p by $\mathcal{Z}(p)$. More generally, for a subset $B \subseteq \mathbb{Z}[x_1, \ldots, x_n]$, $\mathcal{Z}(B) = \{\sigma \mid \forall p \in B : p^\sigma = 0\}$. For later use, let us state some facts concerning zeros, in particular of the relationship of operations on ideals with operations on their zeros.

Lemma 2. *Suppose B, B' are sets of polynomials, q is a polynomial, I, I' are ideals, and $\mathcal{I}$ is a set of ideals in $\mathbb{Z}[x_1, \ldots, x_n]$.*

1. *If $B \subseteq B'$ then $\mathcal{Z}(B) \supseteq \mathcal{Z}(B')$.*
2. *$\mathcal{Z}(B) = \mathcal{Z}((B)) = \bigcap_{p \in B} \mathcal{Z}(p)$. In particular, $\mathcal{Z}(q) = \mathcal{Z}((q))$.*
3. *$\mathcal{Z}(\sum \mathcal{I}) = \bigcap \{\mathcal{Z}(I) \mid I \in \mathcal{I}\}$. In particular, $\mathcal{Z}(I + I') = \mathcal{Z}(I) \cap \mathcal{Z}(I')$.*
4. *$\mathcal{Z}(\bigcap \mathcal{I}) = \bigcup \{\mathcal{Z}(I) \mid I \in \mathcal{I}\}$, if $\mathcal{I}$ is finite. In particular, $\mathcal{Z}(I \cap I') = \mathcal{Z}(I) \cup \mathcal{Z}(I')$.*
5. *$\mathcal{Z}(\{0\}) = \Sigma$ and $\mathcal{Z}(\mathbb{Z}[x_1, \ldots, x_n]) = \emptyset$.*
6. *$\mathcal{Z}(I) = \Sigma$ if and only if $I = \{0\} = (0)$.*

Substitution. Suppose given a polynomial $p \in \mathbb{Z}[x_1, \ldots, x_n]$ and a variable $x \in X$. We can define a substitution operation on ideals I as follows: $I[p/x] = (\{q[p/x] \mid q \in I\})$, where the substitution of polynomial p for x in q, $q[p/x]$, is defined as usual. By definition, $I[p/x]$ is the smallest ideal that contains all polynomials $q[p/x]$ with $q \in I$. From a basis for I, a basis for $I[p/x]$ is obtained in the expected way: if $I = (B)$, then $I[p/x] = (\{b[p/x] \mid b \in B\})$. Thus, we can easily obtain a finite basis for $I[p/x]$ from a finite basis for I: if $I = (b_1, \ldots, b_k)$ then $I[p/x] = (b_1[p/x], \ldots, b_k[p/x])$. Hence we can add substitution to our list of computable operations.

The substitution operation on ideals defined in the previous paragraph mirrors precisely semantic substitution in assertions which has been defined in connection with $\mathsf{wp}(x := e)$.

Lemma 3. $\mathcal{Z}(I)[p/x] = \mathcal{Z}(I[p/x])$.

We leave the proof of this equation that involves the substitution lemma known from logic to the reader.

Projection. In this section we define projection operators $proj_i$, $i = 1, \ldots, n$, such that for each ideal I, $\mathcal{Z}(proj_i(I)) = \forall x_i(\mathcal{Z}(I))$. Semantic universal quantification over assertions has been defined in connection with $\mathsf{wp}(x :=?)$.

A polynomial $p \in \mathbb{Z}[x_1, \ldots, x_n]$ can uniquely be written as a polynomial in x_i with coefficients in $\mathbb{Z}[x_1, \ldots, x_{i-1}, x_{i+1}, x_n]$, i.e., in the form $p = c_k x_i^k + \ldots + c_0 x_i^0$, where $c_0, \ldots, c_k \in \mathbb{Z}[x_1, \ldots, x_{i-1}, x_{i+1}, x_n]$, and $c_k \neq 0$ if $k > 0$. We call $c_0, \ldots, c_k$ the coefficients of p with respect to x_i and let $\mathcal{C}_i(p) = \{c_0, \ldots, c_k\}$.

Lemma 4. $\forall x_i(\mathcal{Z}(p)) = \mathcal{Z}(\mathcal{C}_i(p))$.

Proof. Let $p = c_k x_i^k + \ldots + c_0 x_i^0$ with $\mathcal{C}_i(p) = \{c_0, \ldots, c_k\}$.

'$\supseteq$': Let $\sigma \in \mathcal{Z}(\mathcal{C}_i(p))$. We have $c_k^{\sigma[x_i \mapsto d]} = c_k^{\sigma} = 0$ for all $d \in \mathbb{Z}$ because c_k is independent of x_i. Hence, $p^{\sigma[x_i \mapsto d]} = c_k^{\sigma[x_i \mapsto d]} d^k + \ldots + c_0^{\sigma[x_i \mapsto d]} d^0 = 0d^k + \ldots + 0d^0 = 0$ for all $d \in \mathbb{Z}$, i.e. $\sigma \in \forall x_i(\mathcal{Z}(p))$.
'$\subseteq$': Let $\sigma \in \forall x_i(\mathcal{Z}(p))$. We have $c_k^{\sigma[x_i \mapsto d]} = c_k^{\sigma}$ for all $d \in \mathbb{Z}$ because c_k is independent of x_i. Therefore, $c_k^{\sigma} d^k + \ldots + c_0^{\sigma} d^0 = c_k^{\sigma[x_i \mapsto d]} d^k + \ldots + c_0^{\sigma[x_i \mapsto d]} d^0 = p^{\sigma[x_i \mapsto d]} = 0$ for all $d \in \mathbb{Z}$ because of $\sigma \in \forall x_i(\mathcal{Z}(p))$. This means that the polynomial $c_k^{\sigma} x_i^k + \ldots + c_0^{\sigma} x_i^0$ vanishes for all values of x_i. Hence, it has more than k zeros which implies that it is the zero polynomial. Consequently, $c_j^{\sigma} = 0$ for all $j = 0, \ldots, k$, i.e., $\sigma \in \mathcal{Z}(\mathcal{C}_i(p))$. □

Suppose $I \subseteq \mathbb{Z}[x_1, \ldots, x_n]$ is an ideal with basis B.

Lemma 5. $\forall x_i(\mathcal{Z}(I)) = \mathcal{Z}(\bigcup_{p \in B} \mathcal{C}_i(p))$.

Proof. $\forall x_i(\mathcal{Z}(I)) = \forall x_i(\mathcal{Z}(B)) = \forall x_i(\bigcap_{p \in B} \mathcal{Z}(p)) = \bigcap_{p \in B} \forall x_i(\mathcal{Z}(p)) = \bigcap_{p \in B} \mathcal{Z}(\mathcal{C}_i(p)) = \mathcal{Z}(\bigcup_{p \in B} \mathcal{C}_i(p))$. □

In view of this formula, it is natural to define $proj_i(I) = (\bigcup_{p \in B} \mathcal{C}_i(p))$ where B is a basis of I. It is not hard to show that this definition is independent of the basis; we leave this proof to the reader. Obviously, $proj_i$ is effective: if I is given by a finite basis $\{b_1, \ldots, b_k\}$ then $proj_i(I)$ is given by the finite basis $\bigcup_{j=1}^{k} \mathcal{C}_i(b_j)$.

Corollary 1. $\forall x_i(\mathcal{Z}(I)) = \mathcal{Z}(proj_i(I))$.

Proof. $\forall x_i(\mathcal{Z}(I)) = \mathcal{Z}(\bigcup_{p \in B} \mathcal{C}_i(p)) = \mathcal{Z}((\bigcup_{p \in B} \mathcal{C}_i(p))) = \mathcal{Z}(proj_i(I))$. □

7 Detection of Polynomial Constants

We represent assertions by ideals of the polynomial ring $\mathbb{Z}[x_1, \ldots, x_n]$ in the detection algorithm for polynomial constants. Thus, let $\mathbb{D}$ be the set of ideals of $\mathbb{Z}[x_1, \ldots, x_n]$) and $\sqsubseteq$ be $\supseteq$. The representation mapping is $\gamma(I) = \mathcal{Z}(I)$. Note that the order is *reverse* inclusion of ideals. This is because larger ideals have smaller sets of zeros. Thus, the *meet* operation is the *sum* operation of ideals and the top element is the ideal $\{0\} = (0)$.

In a practical algorithm, ideals are represented by finite bases. For transparency, we suppress this further representation step but ensure that only operations that can effectively be computed on bases are used.

The lattice $(\mathbb{D}, \supseteq)$ satisfies requirements (a)–(g) of Section 4:

(a) $\mathbb{Z}[x_1, \ldots, x_n]$ is Noetherian.
(b) By the identity $\mathcal{Z}(\sum \mathcal{I}) = \bigcap\{\mathcal{Z}(I) \mid I \in \mathcal{I}\}$, $\mathcal{Z}$ is universally conjunctive.
(c) Suppose $x \in X$ and $c \in \mathbb{Z}$. Certainly, a state is a zero of the ideal generated by the polynomial $x - c$ if and only if it maps x to c. Hence, we choose $d_{x=c}$ as the ideal $(x - c)$ generated by $x - c$.
(d) In Section 5 we have seen that the sum of two ideals can effectively be computed on bases.
(e) By Section 6, $(\cdot)[p/x]$ is an adequate, computable substitution operation.

(f) Again by Section 6, $proj_i$ is an adequate, computable projection operation.
(g) We know that $\mathcal{Z}(I) = \Sigma$ if and only if $I = \{0\}$. Moreover, the only basis of the ideal $\{0\}$ is $\{0\}$ itself. Hence, in order to decide whether an ideal I given by a basis B represents Σ, we only need to check whether $B = \{0\}$.

We can thus apply the generic algorithm from Section 4 for the detection of polynomial constants. In order to make this more specific, we put the pieces together, and describe the resulting algorithm in more detail.

Suppose given a variable $x \in X$ and a program point $w \in N$ in a flow graph $G = (N, E, A, \mathbf{s}, \mathbf{e})$. Then the following algorithm decides whether x is a polynomial constant at w or not:

Phase 1: Determine a candidate value $c \in \mathbb{Z}$ for x at w by executing an arbitrary (cycle-free) path from $\mathbf{s}$ to w.

Phase 2: Associate with each edge $(u, v) \in E$ a transfer function $f_{(u,v)} : \mathbb{D} \to \mathbb{D}$ that represents $\mathsf{wp}(A_S(u, v))$:

$$f_{(u,v)}(I) = \begin{cases} I & \text{if } A(u,v) = \mathsf{skip} \\ I[p/x] & \text{if } A(u,v) = (x := p) \text{ with } p \in \mathbb{Z}[x_1, \ldots, x_n] \\ proj_x(I) & \text{if } A(u,v) = (x := t) \text{ with } t \notin \mathbb{Z}[x_1, \ldots, x_n] \\ proj_x(I) & \text{if } A(u,v) = (x :=?) \end{cases}$$

Set $A_0[w] = (x - c)$ and $A_0[u] = (0)$ for all $u \in N \backslash \{w\}$ and compute the largest solution (w.r.t. $\sqsubseteq = \supseteq$) of the equation system

$$\mathbf{A}[u] = A_0[u] + \sum_{v \in Succ[u]} f_{(u,v)}(\mathbf{A}[v]) \quad \text{for each } u \in N\,.$$

We can do this as follows. Starting from $A_0[u]$ we iteratively compute, simultaneously for all program points $u \in N$, the following sequences of ideals

$$A_{i+1}[u] = A_i[u] + \sum_{v \in Succ[u]} f_{(u,v)}(\mathbf{A}_i[v])\,.$$

We stop upon stabilization, i.e., when we encounter an index i_s with $A_{i_s+1}[u] = A_{i_s}[u]$ for all $u \in N$. Obviously, $A_0[u] \subseteq A_1[u] \subseteq A_2[u] \subseteq \ldots$, such that computation must terminate eventually because $\mathbb{Z}[x_1, \ldots, x_n]$ is Noetherian. In this computation we represent ideals by finite bases and perform Gröbner basis computations in order to check whether $A_{i+1}[u] = A_i[u]$.[2]

Phase 3: Check if the ideal computed for the start node, $A_{i_s}[\mathbf{s}]$, is (0). If so, x is a polynomial constant of value v at w; otherwise, x is not a polynomial constant at w.

Phase 2 can be seen as a backwards data flow analysis in a framework in which ideals of $\mathbb{Z}[x_1, \ldots, x_n]$ constitute data flow facts, the transfer functions

[2] As $A_{i+1}[u] \supseteq A_i[u]$ by construction, it suffices to check $A_{i+1}[u] \subseteq A_i[u]$.

are the functions $f_{(u,v)}$ specified above, and the start value is A_0. Of course, we can use any evaluation strategy instead of naive iteration.

We do not know any complexity bound for our algorithm. Our termination proof relies on Hilbert's basis theorem and its standard proof is non-constructive and does not provide an upper bound for the maximal length of strictly increasing chains of ideals. Therefore, we cannot bound the number of iterations performed by our algorithm.

8 Conclusion

In this paper we have shown that polynomial constants are decidable. Our algorithm can easily be extended to handle conditions of the form $p \mathrel{/=} 0$ with $p \in \mathbb{Z}[x_1, \ldots, x_n]$. The weakest precondition is $\mathsf{wp}(p \mathrel{/=} 0, \phi) = (p \mathrel{/=} 0 \Rightarrow \phi) = (p = 0 \vee \phi)$ and if ϕ is represented by an ideal I, the assertion $p = 0 \vee \phi$ is represented by the ideal $I \cap (p)$ according to Lemma 2. This observation can be used to handle such conditions in our algorithm. We can extend this easily to an arbitrary mixture of disjunctions and conjunctions of conditions of the form $p \mathrel{/=} 0$. Of course, we cannot handle the dual form of conditions $p = 0$: with both types of conditions we can obviously simulate two-counter machines.

The idea to detect constants with a symbolic weakest precondition computation has previously been used in a polynomial-time algorithm for detection of *Presburger constants* [16]. In Presburger constant detection only the integer operators $+$ and $-$ are interpreted and assertions are represented by affine vector spaces over $\mathbb{Q}$. In contrast to our algorithm, the Presburger constant detection algorithm cannot easily be extended to conditions as affine spaces are not closed under union.

Standard constant propagation relies on forward propagation while we use backwards propagation of assertions. Interestingly, Presburger constants can also be detected by forward propagation of affine spaces. Karr [10] describes such an algorithm but does not address completeness issues. In forward propagation of assertions we effectively compute strongest postconditions rather than weakest precondition and this computation involves union of assertions rather than intersection. Because affine spaces are not closed under union, Karr defines a (complicated) union operator of affine spaces that over-approximates their actual union by an affine space. One is tempted to consider forward propagation of ideals of $\mathbb{Z}[x_1, \ldots, x_n]$. At first glance, this idea looks promising, because ideals are closed under intersection and intersection of ideals corresponds to union of their sets of zeros, such that we can even precisely represent the union of assertions. There is, however, another problem: $\mathbb{Z}[x_1, \ldots, x_n]$ is not 'co-Noetherian', i.e., there are infinitely long strictly *decreasing* chains of ideals, e.g., $(x) \supset (x^2) \supset (x^3) \supset \ldots$. Therefore, strongest postcondition computations with ideals cannot be guaranteed to terminate in general.

Our approach to compute weakest preconditions symbolically with effective representations is closely related to abstract interpretation [3,4]. Requirement (b) of the generic algorithm, universal conjunctivity of the representation mapping

$\gamma : \mathbb{D} \to 2^{\Sigma}$, implies that γ has a lower adjoint, i.e., that there is a monotonic mapping $\alpha : 2^{\Sigma} \to \mathbb{D}$ such that (α, γ) is a Galois connection [13]. In the standard abstract interpretation framework, we are interested in computation of least fixpoints and the lower adjoint, α, is the abstraction mapping. Here, we are in the dual situation: we are interested in computation of greatest fixpoints. Thus, the role of the abstraction is played by the upper adjoint, $\gamma : \mathbb{D} \to 2^{\Sigma}$. Funnily, this means that in a technical sense the members of $\mathbb{D}$ provide more concrete information than the members of 2^{Σ} and that we compute on the concrete side of the abstract interpretation. Thus, we interpret the lattice $\mathbb{D}$ as an *exact partial representation* rather than an abstract interpretation. The representation via $\mathbb{D}$ is *partial* because it does not represent all assertions exactly; this is indispensable due to countability reasons because we cannot represent all assertions effectively. It is an *exact representation* because it allows us to infer the weakest preconditions arising in the S-constant algorithms precisely, which is achieved by ensuring that the initial value of the fixpoint computation is represented exactly and that the occurring operations on representations mirror the corresponding operations on assertions precisely.

By the very nature of Galois connections, the representation mapping γ and its lower adjoint α satisfy the two inequalities $\alpha \circ \gamma \sqsubseteq \mathsf{Id}_{\mathbb{D}}$ and $\mathsf{Id}_{2^{\Sigma}} \subseteq \gamma \circ \alpha$, where $\mathsf{Id}_{\mathbb{D}}$ and $\mathsf{Id}_{2^{\Sigma}}$ are the identities on $\mathbb{D}$ and 2^{Σ}, respectively. Interestingly, none of these inequalities degenerates to an equality when we represent assertions by ideals of $\mathbb{Z}[x_1, \ldots, x_n]$ as in our algorithm for detection of polynomial constants. On the one hand, $\gamma \circ \alpha \not= \mathsf{Id}_{2^{\Sigma}}$ because the representation is necessarily partial. On the other hand, $\alpha \circ \gamma \not= \mathsf{Id}_{\mathbb{D}}$ because the representation of assertions is not unique. For example, if $p \in \mathbb{Z}[x_1, \ldots, x_n]$ does not have a zero in the integers, we have $\mathcal{Z}((p)) = \emptyset$ such that $\mathcal{Z}((p)) = \mathcal{Z}((1)) = \mathcal{Z}(\mathbb{Z}[x_1, \ldots, x_n])$. But by undecidability of Hilbert's tenth problem, we cannot decide whether we are faced with such a polynomial p and thus cannot effectively identify (p) and (1). This forces us to work with a non-unique representation. While we cannot decide whether the set of zeros of an ideal I given by a basis B is empty, we can decide whether it equals Σ because this only holds for $I = (0)$. Fortunately, this is the only question that needs to be answered for the weakest precondition.

As a consequence of non-uniqueness, the weakest precondition computation on ideals does not necessarily stop once it has found a collection of ideals that represents the largest fixpoint on assertions but may proceed to larger ideals that represent the same assertions. Fortunately, we can still prove termination by arguing on ideals directly.

Let us discuss some possible directions for future work. Firstly, it is interesting to implement the detection algorithm for polynomial constants and evaluate how it performs in practice. Secondly, we can look for other applications of the generic algorithm. Of course, we can tackle, e.g, polynomial constants over $\mathbb{Q}$ rather than $\mathbb{Z}$, where we can use essentially the same algorithm because $\mathbb{Q}[x_1, \ldots, x_n]$ is also a strongly computable ring. But we may also identify other classes where assertions can be represented symbolically. On the theoretical side, there is the challenge to diminish the gap between the upper and lower complexity bound for the detection problem of polynomial constants. Currently, we have decidability

as an upper bound, as witnessed by the algorithm in this paper, and PSPACE-hardness as a lower bound [16].

Acknowledgment. We thank the anonymous referees for their comments that helped to improve the submitted version.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, 1986.
3. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings 4th POPL*, Los Angeles, California, 1977.
4. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic Computat.*, 4(2):511–547, 1992.
5. J. H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra: Systems and Algorithms for Algebraic Computation.* Academic Press, 1988.
6. E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.
7. C. Fischer and R. LeBlanc. *Crafting a Compiler.* Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA, 1988.
8. K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra.* Kluwer, 1992.
9. M. S. Hecht. *Flow analysis of computer programs.* Elsevier North-Holland, 1977.
10. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.
11. Mathematics of Program Construction Group. Fixed-point calculus. *Information Processing Letters*, 53(3):131–136, 1995.
12. Y. V. Matiyasevich. *Hilbert's Tenth Problem.* The MIT Press, 1993.
13. A. Melton, D. A. Schmidt, and G. E. Strecker. Galois connections and computer science applications. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, editors, *Category Theory and Computer Programming*, LNCS 240, pages 299–312. Springer-Verlag, 1985.
14. B. Mishra. *Algorithmic Algebra.* Springer-Verlag, 1993.
15. S. S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, San Francisco, California, 1997.
16. M. Müller-Olm and O. Rüthing. The complexity of constant propagation. In D. Sands, editor, *ESOP 2001*, LNCS 2028, pages 190–205. Springer, 2001.
17. J. R. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. In *Conf. Rec. 4th ACM Symposium on Principles of Programming Languages POPL'77*, pages 104–118, Los Angeles, CA, January 1977.
18. M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.
19. B. Steffen and J. Knoop. Finite constants: Characterizations of a new decidable set of constants. *Theoretical Computer Science*, 80(2):303–318, 1991.
20. F. Winkler. *Polynomial Algorithms.* Springer-Verlag, 1996.

Normalizable Horn Clauses, Strongly Recognizable Relations, and Spi

Flemming Nielson[1], Hanne Riis Nielson[1], and Helmut Seidl[2]

[1] Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Kongens Lyngby, Denmark, {nielson,riis}@imm.dtu.dk
[2] Universität Trier, FB IV – Informatik, D-54286 Trier, Germany, seidl@uni-trier.de

Abstract. We exhibit a rich class of Horn clauses, which we call $\mathcal{H}1$, whose least models, though possibly infinite, can be computed effectively. We show that the least model of an $\mathcal{H}1$ clause consists of so-called *strongly* recognizable relations and present an exponential normalization procedure to compute it. In order to obtain a practical tool for program analysis, we identify a restriction of $\mathcal{H}1$ clauses, which we call $\mathcal{H}2$, where the least models can be computed in polynomial time. This fragment still allows to express, e.g., Cartesian product and transitive closure of relations. Inside $\mathcal{H}2$, we exhibit a fragment $\mathcal{H}3$ where normalization is even cubic. We demonstrate the usefulness of our approach by deriving a cubic control-flow analysis for the Spi calculus [1] as presented in [14].

Keywords: Program analysis, uniform Horn clauses, strongly recognizable relations, Spi calculus.

1 Introduction

In [16], we proposed ALFP ("alternation-free least fixpoint logic in clausal form") as a specification language for control-flow analyzers. ALFP extends classical Datalog clauses by extra logical features like, e.g., explicit quantification, disjunctions in pre-conditions and conditional clauses. ALFP and related formalisms have shown to be extremely convenient for specifying control-flow analyses for various programming languages [11,15]. The advantage of such a logic-based approach is that tuning of the analysis boils down to rewriting of formulas whose correctness can be formally proven within the logic. Also, the generic implementation of formula solvers gives rise to complexity meta-theorems [3,11,7] thus forming the basis for generators of *predictable program analyses* [16,13]. The ALFP based approach as well as the restricted classes of Horn clauses used by McAllester in [11], though, have the draw-back that the result of the analysis, i.e., the least model of the specification is necessarily finite.

Here, we try to lift this limitation. A classical approach uses (classes of) set constraints [8,4,17,2]. As a more flexible formalism, Frühwirth et al. proposed a syntactical restriction of Horn clauses which they called *uniform*, guaranteeing that the least model consists of *recognizable sets* of trees and, moreover, can be effectively computed [6]. Recall that a set of trees is *recognizable* iff it can be

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 20–35, 2002.
© Springer-Verlag Berlin Heidelberg 2002

accepted by a finite tree automaton. In essence, Frühwirth et al. consider unary predicates only and restrict implications to be of one of the following forms:

$$\begin{array}{lcl} p(X) & \Leftarrow & p_1(t_1), \ldots, p_m(t_m) \\ p(a) & \Leftarrow & p_1(t_1), \ldots, p_m(t_m) \\ p(t) & \Leftarrow & p_1(X_1), \ldots, p_n(X_n) \end{array}$$

where a is an atom, t is linear, and t_i are arbitrary terms. So, the implications

$$\begin{array}{lcll} p(a(X,Y)) & \Leftarrow & q(b(X,Y)) & \text{(constructor renaming)} \\ p(a(X,Z)) & \Leftarrow & q_1(a(X,Y)), q_2(a(Y,Z)) & \text{(composition)} \end{array}$$

are *not* uniform. Uniform Horn clauses (like set constraints) are *too weak* since they are not well suited for dealing with *relations*. Neither do they conveniently support Cartesian product, nor transitive closure or projections onto more than just one component. On the other hand, set constraints (like uniform Horn clauses) in general are *too strong* a formalism since computing their least solutions easily becomes exponential-time hard [6,18,4].

Here, we follow Frühwirth et al. in [6] by using (subclasses of) ordinary Horn clauses as specifications of analyses and *generalize* it by allowing non-unary predicates. Clearly, without any restrictions the resulting relations will be neither finite nor effectively computable. In order to obtain effective descriptions for relations, we restrict ourselves to *strongly recognizable* relations which we define to be *finite unions of Cartesian products* of recognizable tree sets. We exhibit a rich class of Horn clauses where the least model consists of strongly recognizable tree relations only and present an exponential-time normalization procedure to compute least models. This class, which we call $\mathcal{H}1$, allows renaming and composition as defined above. Furthermore, it conveniently supports relational operations like Cartesian product, transitive closure and arbitrary projections.

As $\mathcal{H}1$ can express unrestricted intersection of recognizable tree sets by a formula of polynomial size, computing the least model of $\mathcal{H}1$ clauses is hard for deterministic exponential time [18]. Thus, the class $\mathcal{H}1$ has the same efficiency problems as set constraints or uniform Horn clauses. Therefore, we exhibit in section 4 a large subclass which we call $\mathcal{H}2$. This subclass still supports, e.g., constructor renaming, Cartesian product, transitive closure and arbitrary projections. The complexity of $\mathcal{H}2$ is polynomial: the exponent, however, depends on the number of variables in individual implications. Therefore, we exhibit in section 5 a subclass $\mathcal{H}3$ of $\mathcal{H}2$ for which a uniformly cubic solving time can be proven. The normalization procedure for $\mathcal{H}3$ then is used in section 6 to obtain a control-flow analysis for the Spi calculus [1] as presented in [14] which is cubic.

2 Basics

A Horn clause is a finite set of *implications*. Every implication r is of the form $h \Leftarrow \alpha$ where h and α are the *head* and the *pre-condition* of r, respectively. Every pre-condition equals a sequence of *queries*. The head and every query are *literals*, i.e., of the form $p(t_1, \ldots, t_k)$ where p is a k-ary predicate and t_i are terms which are built up from variables (starting with capital letters) and constants through constructor applications. For convenience, we also introduce the abbreviation: $h_1, \ldots, h_n \Leftarrow \alpha$ for the set of implications:

$$h_1 \Leftarrow \alpha \quad \ldots \quad h_n \Leftarrow \alpha$$

all of which share the same pre-condition α. For a signature Σ of constants and constructors, we denote the set of all finite variable-free (*ground*) terms by T_Σ. A k-ary *tree relation* is a subset of T_Σ^k. Horn clauses are interpreted relative to an interpretation ρ which maps predicate symbols to corresponding tree relations.

Let ρ denote an interpretation of occurring predicates, and θ denote a ground substitution mapping the occurring variables to ground terms. Relative to ρ and θ, we define for literals g, pre-conditions α, implications $h \Leftarrow \alpha$ and Horn clauses c, the *satisfaction* relation "$\models$" by:

$$
\begin{array}{lllll}
(1) & \rho, \theta \models p(t_1, \ldots, t_k) & \text{iff} & \theta(t_1, \ldots, t_k) \in \rho(p) \\
(2) & \rho, \theta \models g_1, \ldots, g_m & \text{iff} & \forall\, j:\ \rho, \theta \models g_j \\
(3) & \rho \quad \models h \Leftarrow \alpha & \text{iff} & \forall\, \theta:\ \rho, \theta \models \alpha \ \Rightarrow\ \rho, \theta \models h \\
(4) & \rho \quad \models c & \text{iff} & \rho \models r \text{ for all implications } r \text{ in } c
\end{array}
$$

An interpretation ρ with $\rho \models c$ is also called *model* of c. It is well-known that the set of models of a Horn clause forms a Moore family. Therefore, there is a unique least model.

3 Normalizable Horn Clauses

Unrestricted Horn clauses have Turing power. In order to arrive at a decidable fragment, we have to impose restrictions. Horn clauses define tree relations. A k-ary tree relation R is *recognizable* iff the set of tuples $(t_1, \ldots, t_k) \in R$ can be characterized by a finite tree automaton running on all k trees simultaneously. Recognizable tree relations have been introduced by Lugiez and Schnoebelen for model-checking of PA processes and deciding first-order transition logic [10]. Recognizable relations enjoy many useful closure properties. In particular, they contain all finite relations and are closed under boolean operations, projection, Cartesian product and composition. They are in general not closed, though, under transitive closure.

Therefore, we consider a subclass of recognizable tree relations only. We call a tree relation $R \subseteq T_\Sigma^k$ *strongly recognizable* iff R equals a finite union of k-fold Cartesian products of recognizable sets of trees. In particular, unary strongly recognizable tree relations simply are recognizable *sets* of trees. Not every recognizable tree relation, though, is also strongly recognizable. The simplest counter example is the identity relation which is trivially recognizable but not strongly recognizable (unless the universe is finite). Like recognizable relations, strongly recognizable relations are closed under boolean operations, projection, Cartesian product and composition (see chapter 3, section 2 of [5]). As a subclass, strongly recognizable relations enjoy stronger closure properties than general recognizable relations. Here, we show that the transitive closure of a binary strongly recognizable relation is again strongly recognizable (see corollary 1 below). In terms of program analysis, the so-called *independent attribute method* [9] is often taken to be a Cartesian product of powersets of values whereas the so-called *relational method* [9] often is taken to be a powerset of a Cartesian product of values; in this sense, our notion of *strongly recognizable relations* constitutes a "finitary relational method" that is more versatile than the independent attribute method and specializes to the relational method in the case of a finite universe.

We call an implication *normal* iff it is of one of the forms:

$$
\begin{array}{lll}
(N1) & p(b) & \Leftarrow \\
(N2) & p(f(X_1,\ldots,X_k)) & \Leftarrow \quad q_1(X_1),\ldots,q_k(X_k) \\
(N3) & p(X_1,\ldots,X_k) & \Leftarrow \quad q_1(X_1),\ldots,q_k(X_k)
\end{array}
$$

for distinct variables X_i. In the following, we subsume the first case into the second one by allowing also constructors f of arity 0. Also, we call a Horn clause c *normal* iff every implication occurring in c is normal. Every strongly recognizable relation can be described through a normal Horn clause. In fact, the reverse is also true:

Proposition 1. *Assume c is a normal Horn clause. Then the least model of c maps every occurring predicate to a strongly recognizable relation.* □

We call a Horn clause c *normalizable* iff a normal Horn clause c' can be constructed from c which is equivalent to c (up to auxiliary relations). We are interested in large classes of normalizable Horn clauses in order to allow as much flexibility as possible when specifying analyses. In order to characterize such classes, we introduce the following notion. To a sequence α of queries, we associate the (undirected) *variable dependence graph* $D_\alpha = (V_\alpha, E_\alpha)$. The set of vertices of D_α equals the set of queries g occurring in α, whereas the set of edges is given by $\{g_1, g_2\} \in E_\alpha$ iff $g_1 \not\models g_2$ and $\mathit{Vars}(g_1) \cap \mathit{Vars}(g_2) \not\models \emptyset$. Two variables X_1, X_2 are *connected* (w.r.t. α) iff they occur within queries which are connected in D_α. In particular, variables within the same query are connected.

The implication $h \Leftarrow \alpha$ has property H1 iff

(1) h is linear, i.e., no variable occurs twice in h.
(2) If two variables X, Y in h are connected, then X and Y are siblings in h.

Here, we call two variables *siblings* in a literal or term if they occur as arguments of a common father. So, X, Y are siblings in $p(X,Y)$ and in $p(a(X,b,Y))$ but not siblings in $p(X, a(Y))$. A Horn clause c belongs to the class $\mathcal{H}1$ iff all implications in c are H1.

For the following, we fix a finite set Σ of constructors. In particular, the maximal arity $a[c]$ of a constructor or predicate in the clause c is non-zero but $\mathcal{O}(1)$. We *assume* in our complexity estimations that the maximal number of queries as well as the maximal number $v[c]$ of variable occurrences in a single implication is non-zero and $\mathcal{O}(1)$. Note that this does neither preclude the *number* of implications of c nor the *sizes* of occurring ground subterms to be unbounded. The maximum of 2 and the maximal number of repetitions of a variable in a pre-condition of c is denoted by $r[c]$. Also, we will refer to the *size* $|c|$ of c which we define as the size of a "natural" internal representation of c. For this representation, we use a condensed version of the abstract syntax tree of the clause where ground subterms are maximally shared. We have:

Theorem 1. *For every clause from $\mathcal{H}1$, an equivalent normal clause can be constructed in deterministic exponential time.*

Proof (Sketch). We proceed in two steps. In the first step, we transform the $\mathcal{H}1$-clause into an equivalent clause of a particularly simple form which we then normalize in the second step.

A clause from $\mathcal{H}1$ is $\mathcal{H}1$-*special* iff every implication is of one of the forms:

$$(1) \qquad p(X_1, \ldots, X_k) \Leftarrow q_1(\underline{Y}_1), \ldots, q_m(\underline{Y}_m)$$

$$(2) \qquad p(f(X_1, \ldots, X_k)) \Leftarrow q_1(\underline{Y}_1), \ldots, q_m(\underline{Y}_m)$$

$$(3) \qquad p(X_1, \ldots, X_k) \Leftarrow q(Y_1, \ldots, Y_{j-1}, f(Z_1, \ldots, Z_{n'}), Y_{j+1}, \ldots, Y_n), \mathbf{p}$$

where the variables X_i are pairwise distinct, the $\underline{Y}_i$ are arbitrary sequences of variables, and $\mathbf{p}$ is a sequence of unary queries. We have:

Proposition 2. *For a Horn clause c from $\mathcal{H}1$, a $\mathcal{H}1$-special Horn clause c' can be constructed in linear time which is equivalent to c (up to auxiliary predicates) such that c' has the following properties:*

1. $a[c'] \leq v[c] + a[c] - 1$ *and* $v[c'] \leq v[c] + a[c] - 1$;
2. $r[c'] \leq r[c]$.

Thus, the transformation from c to c' only moderately increases the number of variable occurrences per implication while not increasing the number of variable repetitions. Here, we do not provide a formal proof of prop. 2. Instead, we illustrate the two crucial steps by examples. First, we need a (linear time) transformation which produces an equivalent Horn clause such that all occurring heads are of one of the following forms:

$$p(X_1, \ldots, X_k) \qquad \text{or} \qquad p(f(X_1, \ldots, X_k))$$

Example 1. Consider the implication: $p(a(X, Y, b), c(Z)) \Leftarrow \alpha$.

By definition of H1, neither X and Z nor Y and Z can be connected. Therefore, we can split α into two sequences α_1, α_2 of queries where α_1 contains all queries containing variables which are connected to X or Y and α_2 contains the remaining ones (in particular, those constraining Z). We replace the implication equivalently with the following four implications:

$$\begin{array}{lcl@{\qquad\qquad}lcl} p(X_1, X_2) & \Leftarrow & p_1(X_1), p_2(X_2) & h(b) & \Leftarrow & \\ p_1(a(X, Y, Z)) & \Leftarrow & h(Z), \alpha_1 & p_2(c(Z)) & \Leftarrow & \alpha_2 \end{array}$$

The first implication separates the implication into two queries, one for each argument of p. The following two implications construct the first parameter of p. Each of these create one constructor in the head. The second implication corresponds to the application of a. The bindings for X and Y are provided by α_1, whereas Z is a new auxiliary variable which receives the ground term b through the query to the additional auxiliary predicate h. Finally, the second parameter of p is provided through the fourth implication. □

Next, we simulate complex queries by simpler ones.

Example 2. Consider the complex query $g \equiv p(a(c(X), X), d(Y))$ occurring in some pre-condition. Then g is replaced with the query $p_0(X, X, Y)$ where p_0 is a new predicate with one parameter position for every variable occurrence of g. The predicate p_0 is then defined through the following traversal over g:

$$\begin{array}{lcl} p_0(X_1, X_2, X_3) & \Leftarrow & p_1(c(X_1), X_2, X_3) \\ p_1(X_1, X_2, X_3) & \Leftarrow & p_2(a(X_1, X_2), X_3) \\ p_2(X_1, X_2) & \Leftarrow & p(X_1, d(X_2)) \end{array}$$

Each of the defining implications corresponds to one constructor occurrence in g. The implication for p_1, e.g., inverts the constructor occurrence of a. □

According to proposition 2, it suffices to normalize $\mathcal{H}1$-special clauses only. Assume that c is $\mathcal{H}1$-special. For every subset S of unary predicates, we introduce a new auxiliary unary predicate m_S. In particular, we rename the unary predicates p with $m_{\{p\}}$. The predicate m_S denotes the conjunction of the predicates $p \in S$. The predicate $m_\emptyset$ denotes the empty intersection, i.e., should be true for all trees. For $S = \emptyset$ and every f of rank k, we hence add the rule:

$$m_\emptyset(f(X_1, \ldots, X_k)) \quad \Leftarrow \quad m_\emptyset(X_1), \ldots, m_\emptyset(X_k)$$

By adding queries to $m_\emptyset$, we may w.l.o.g. assume that all head variables also occur in the corresponding pre-conditions.

The general idea of the normalization procedure is to successively add simpler implications until no further simplifications can be applied. Two properties must be enforced. First, the newly added implications should always be implied by the already existing ones. Second, after saturation, all non-normal implications should become superfluous. In the following, we collect the rules for adding new implications. Assume that $\mathcal{H}$ is the least model of the currently obtained set of *normal* implications. Note that it will grow monotonically as clauses are added.

First, consider *chain rules*, i.e., implications of the form:

$$m_S(X) \quad \Leftarrow \quad m_{S'}(X)$$

Then for every normal implication:

$$m_{S'}(f(X_1, \ldots, X_k)) \quad \Leftarrow \quad m_{S_1}(X_1), \ldots, m_{S_k}(X_k)$$

we add the implication:

$$m_S(f(X_1, \ldots, X_k)) \quad \Leftarrow \quad m_{S_1}(X_1), \ldots, m_{S_k}(X_k)$$

If we are given normal implications:

$$m_{\{p\}}(f(X_1, \ldots, X_k)) \quad \Leftarrow \quad m_{S_{p,1}}(X_1), \ldots, m_{S_{p,k}}(X_k)$$

for all $p \in S$, we add the normal implication:

$$m_S(f(X_1, \ldots, X_k)) \quad \Leftarrow \quad m_{S_1}(X_1), \ldots, m_{S_k}(X_k)$$

where $S_j = \bigcup\{S_{p,j} \mid p \in S\}$, i.e., S_j collects all predicates constraining X_j.

Next, assume that the implication is of form (1) or (2) and contains a non-unary query, i.e., equals

$$h \quad \Leftarrow \quad \mathbf{p}_1, q(Y_1, \ldots, Y_n), \mathbf{p}_2$$

($n \neq 1$) where$h$ either equals $p(X_1, \ldots, X_k)$ or $p(f(X_1, \ldots, X_k))$. If q is nullary, i.e., $n = 0$, then we add the implication:

$$h \quad \Leftarrow \quad \mathbf{p}_1, \mathbf{p}_2$$

whenever $\mathcal{H}(q)$ is non-empty. If q has arity at least 2, then for every implication:

$$q(X_1, \ldots, X_n) \quad \Leftarrow \quad m_{S_1}(X_1), \ldots, m_{S_n}(X_n)$$

we add the implication:

$$h \Leftarrow \mathbf{p}_1, m_{S_1}(Y_1), \ldots, m_{S_n}(Y_n), \mathbf{p}_2$$

Now assume that the pre-condition of an implication consists of unary queries only, i.e., the implication equals $h \Leftarrow \mathbf{p}$ where $\mathbf{p}$ is a sequence of unary queries. By assumption, all head variables $X_1, \ldots, X_k$ occur in $\mathbf{p}$. We simplify $\mathbf{p}$ in two steps. First, we join together all unary predicates constraining the same variable: If $\mathbf{p}$ is (up to re-ordering) of the form $m_{S_1}(X), \ldots, m_{S_n}(X), \mathbf{p}'$ (X some variable not occurring in $\mathbf{p}'$), we add the implication:

$$h \quad \Leftarrow \quad m_S(X), \mathbf{p}'$$

where $S = S_1 \cup \ldots \cup S_n$.

Next assume that every variable occurs at most once. We aim at removing the non-head variables. Assume that $\mathbf{p}$ is (up to re-ordering) of the form:

$$m_{S'_1}(Y_1), \ldots, m_{S'_l}(Y_l), m_{S_1}(X_1), \ldots, m_{S_k}(X_k)$$

where $Y_1, \ldots, Y_l$ do not occur in the head. If all sets $\mathcal{H}(m_{S'_i})$ are non-empty, then we add the clause:

$$h \quad \Leftarrow \quad m_{S_1}(X_1), \ldots, m_{S_k}(X_k)$$

Finally, we consider non-normal implications of the form (3), i.e.:

$$p(X_1, \ldots, X_k) \Leftarrow q(Y_1, \ldots, Y_{j-1}, f(Z_1, \ldots, Z_{n'}), Y_{j+1}, \ldots, Y_n), \mathbf{p}$$

and assume that the clause contains the normal implications:

$$\begin{array}{lcl} q(X_1, \ldots, X_n) & \Leftarrow & m_{S_1}(X_1), \ldots, m_{S_j}(X_j), \ldots, m_{S_n}(X_n) \\ m_{S_j}(f(X_1, \ldots, X_{n'})) & \Leftarrow & m_{S'_1}(X_1), \ldots, m_{S'_{n'}}(X_{n'}) \end{array}$$

Then we add the implication:

$$\begin{array}{lcl} p(X_1, \ldots, X_k) & \Leftarrow & m_{S_1}(Y_1), \ldots, m_{S_{j-1}}(Y_{j-1}), \\ & & m_{S'_1}(X_1), \ldots, m_{S'_{n'}}(Z_{n'}), m_{S_{j+1}}(Y_{j+1}), \ldots, m_{S_n}(Y_n), \mathbf{p} \end{array}$$

Each newly added rule does not change the least model of the Horn clause, and $\mathcal{H}$ will continue to be a subset of the least model. The number of distinct normal implications is $2^{\mathcal{O}(|c|)}$. We conclude that also the maximal number of normalization steps for $\mathcal{H}1$-special Horn clauses is $2^{\mathcal{O}(|c|)}$. Now, let c_1 denote a clause which is obtained from c by a maximal sequence of normalization steps and c_0 obtained from c_1 by removing the non-normal clauses. We claim that c_0 and c_1 have the same least models. In order to prove that the equivalent normal Horn clause c_0 can be constructed in deterministic exponential time, we observe that each normalization step can be executed in constant time – given an oracle for non-emptiness of occurring unary predicates. Therefore, it remains to prove that the necessary tests for non-emptiness can be executed efficiently enough. For this we, e.g., may maintain a boolean flag b_S for every encountered set S of unary predicates indicating whether the intersection of the predicates in S has already been shown to be non-empty. This book-keeping amounts to overall costs $2^{\mathcal{O}(|c|)}$. Thus, normalization of $\mathcal{H}1$ clauses can be performed in time $2^{\mathcal{O}(|c|)}$. □

Note that our normalization proof is completely straight-forward — quite in contrast to the corresponding proof for uniform Horn clauses in [6] which refers to *two-way tree automata*. As every Horn clause from $\mathcal{H}1$ can be normalized, it might be argued that $\mathcal{H}1$ has the same expressive power for unary predicates as simpler formalisms like finite tree automata, definite set constraints [8,4,17,2] or uniform Horn clauses. From a practical point of view, however, this is not true. $\mathcal{H}1$ is a more convenient specification formalism as it supports manipulation of *relations*. It allows, e.g., to state implications like:

- $p(X,Y) \Leftarrow q(a(X,Y,Z))$
- $p(X,Y,Z) \Leftarrow q(Y,Z,X)$
- $p(X,Z) \Leftarrow q_1(X,Y), q_2(Y,Z)$

On the other hand, arbitrary projections through constructors, permutations of components and composition of relations as in these examples are not expressible through uniform Horn clauses.

Let us finally remark that the the exponential upper complexity bound which we established for the normalization of $\mathcal{H}1$ clauses is the best one can hope to achieve for such a general class. Any fast normalization procedure allows to construct a fast procedure for testing the emptiness of intersections of sequences of recognizable tree languages (given through finite tree automata) — the latter problem, however, is known to be hard for deterministic exponential time [6,18]. Note that this lower bound even holds for the family of $\mathcal{H}1$ clauses where all implications are of constant size (while the number of implications varies).

4 The Class $\mathcal{H}2$

In this section, we try to break the exponential lower complexity bound by exhibiting a large subclass of $\mathcal{H}1$ clauses which can be normalized much faster. Our key idea is to additionally restrict *how* pre-conditions may affect head variables.

We call $h \Leftarrow \alpha$ H2 iff it is H1, i.e., satisfies (1) and (2) above and additionally:

(3) Every variable which occurs in h, occurs in α at most once.

A Horn clause c is $\mathcal{H}2$ iff all implications in c are H2.

Example 3. The implication: $p(X,Y) \Leftarrow q_1(a(X,Z),b), q_2(b(Z)), q_3(c,X)$ is H1 but not H2 as the variable X occurs in the pre-condition twice. On the contrary, some of the implications generated by the flow-logic specification [12, 15] for the Spi calculus, are H2. Let us assume that the program to be analyzed is specified by the abstract syntax tree prog. Then we maintain unary predicates *reach* and *occurs* which collect all reachable sub-programs and all reachable (data) subexpressions, respectively. In particular, we assert the fact: $reach(\mathsf{prog}) \Leftarrow$. In the abstract syntax tree, we use a constructor $\mathsf{decrypt}$ of arity 4 for collecting the four subtrees corresponding to the parts e, x, k and t in the sub-program: **case** e **of** $\{x\}_k\ t$. The rules of the control-flow analysis (including reachability) from [14] for decryption can be formalized as:

$$\begin{array}{lcl} hasValue(X,Y), reach(T) & \Leftarrow & reach(\mathsf{decrypt}(E,X,K,T)), \\ & & hasValue(K,V), \\ & & hasValue(E,\mathsf{enc}(Y,V)) \\ occurs(E), occurs(K) & \Leftarrow & reach(\mathsf{decrypt}(E,X,K,T)) \end{array}$$

Here, enc is another constructor which creates encrypted messages from a content and a key. All these four implications are H2. □

Theorem 2. *For every $\mathcal{H}2$ clause c, an equivalent normal clause c' can be constructed in time $\mathcal{O}(|c|^m)$ where $m = r[c] \cdot (v[c] + a[c])$.*

Proof (Sketch). We proceed as in the proof of theorem 1. First, we bring c into $\mathcal{H}1$-special form. In fact, the resulting clause c' still is $\mathcal{H}2$. Since every head variable occurs in the corresponding pre-condition at most once, we can abandon the use of any auxiliary predicate m_S where S has cardinality > 1 – given that we replace the emptiness checks of $\mathcal{H}(m_S)$ (S a set of unary predicates) during $\mathcal{H}1$-normalization with the emptiness test for: $\bigcap\{\mathcal{H}(p) \mid p \in S\}$. In particular, the occurring sets S to be tested have cardinality at most $r[c]$. Therefore, we can construct for c', an equivalent normal clause c' of size $\mathcal{O}(|c|^v)$ where the number of normalization steps is of order $\mathcal{O}(|c|^{v+1})$ for $v = v[c] + a[c]$. All necessary tests for non-emptiness then amount to costs $\mathcal{O}(|c|^m)$ where $m = r[c] \cdot (v[c] + a[c])$. Hence, normalization can be performed in time $\mathcal{O}(|c|^m + |c|^{v+1}) = \mathcal{O}(|c|^m)$. □

As a corollary, we obtain a closure property of strongly recognizable relations which distinguishes this class from general recognizable relations:

Corollary 1. *The transitive closure of a strongly recognizable binary relation is again strongly recognizable. It can be computed in cubic time.*

Proof. Assume that the strongly recognizable binary relation edge is defined by the normal Horn clause c. Then we can define the transitive closure by:

$$\begin{array}{lcl} \mathsf{trans}(X,Y) & \Leftarrow & \mathsf{edge}(X,Y) \\ \mathsf{trans}(X,Z) & \Leftarrow & \mathsf{edge}(X,Y), \mathsf{trans}(Y,Z) \end{array}$$

The resulting Horn clause falls into the class $\mathcal{H}2$. Therefore, it is normalizable, and the defined binary relation trans is strongly recognizable.

Concerning the complexity, we observe that there are at most $|c|$ implications for the relation edge. Accordingly, the first implication for trans incurs only cost $\mathcal{O}(|c|)$. It remains to consider the second implication. The original clause contains at most $|c|$ unary predicates. Since trans is binary, the normalized clause can contain at most $|c|^2$ implications for trans. Each of the linearly many pre-conditions of edge potentially must be combined with each of the pre-conditions of trans. This results in at most $\mathcal{O}(|c|^3)$ normalization steps. The emptiness of all pairwise intersections of unary predicates can be determined in time $\mathcal{O}(|c|^2)$. We conclude that normalization requires time $\mathcal{O}(|c|^3 + |c|^2) = \mathcal{O}(|c|^3)$. □

5 The Class $\mathcal{H}3$

In light of theorem 2, the class $\mathcal{H}2$ is (theoretically) feasible – given that every implication does not contain too many variables and variable repetitions. In our example analysis for Spi, we used implications which introduce as many as 9 variable occurrences of 6 distinct variables. The resulting exponent ($\geq 9+2+1 = 12$) is way beyond what can honestly be judged as *practically* feasible for non-trivial input clauses. Therefore, we see a need for subclasses of $\mathcal{H}2$ where the

exponent in the runtime estimation of normalization *uniformly* can be bounded by a small constant. Such a class is exhibited in this section.

The idea of this third class of Horn clauses is to allow tree-like variable dependences only. Thus, $h \Leftarrow \alpha$ is called H3 iff (4) and (5) are satisfied:

(4)	h and every literal in α is linear;
(5)	The variable dependence graph for the sequence $h\alpha$ is acyclic where adjacent literals have at most one variable in common.

Hence, every variable occurs at most twice and in different literals. Therefore, variables from h occur in α at most once. Since (4) and (5) imply property H1, we conclude that H3 implies even property H2. A Horn clause c is called $\mathcal{H}3$ iff all implications in c are H3. The clauses from the Spi analysis from example 3, though, do *not* conform with the definition of H3 — as the implication for *hasValue* violates property (5). We will return to this point in the next section.

Theorem 3. *For every $\mathcal{H}3$ clause c, an equivalent normal clause can be constructed in time $\mathcal{O}(|c|^3)$ where the resulting clause is of size $\mathcal{O}(|c|^2)$.*

Proof. As in the proof of theorem 2, we proceed by first transforming the clause into an equivalent clause of simpler form – which then is normalized in the second step. Note that the normal form here and the corresponding normalization rules are quite different from the ones used in the proofs of theorems 1 and 2.

An $\mathcal{H}3$ clause c is $\mathcal{H}3$-special iff every non-normal implication in c is of one of the forms:

$$
\begin{array}{llll}
(1) & p(X_1,\ldots,X_k) & \Leftarrow & q_1(X_1),\ldots,q_k(X_k),\mathbf{q} \\
(2) & p(X_i) & \Leftarrow & q(f(X_1,\ldots,X_k)),\mathbf{p} \\
(3) & p(X_i) & \Leftarrow & q(X_1,\ldots,X_k),\mathbf{p} \\
(4) & p() & \Leftarrow & q(X_1,\ldots,X_k),p_1(X_1),\ldots,p_k(X_k)
\end{array}
$$

where $\mathbf{p}$ in lines (2) and (3) denotes a sequence of unary queries

$$\mathbf{p} \quad \equiv \quad p_1(X_1),\ldots,p_{i-1}(X_{i-1}),p_{i+1}(X_{i+1}),\ldots,p_k(X_k)$$

and $\mathbf{q}$ in line (1) is a (possibly empty) sequence of nullary queries. We have:

Proposition 3. *For a Horn clause c from $\mathcal{H}3$, a $\mathcal{H}3$-special Horn clause c' can be constructed in linear time which is equivalent to c (up to auxiliary predicates).*

Proof. In order to decompose an arbitrary $\mathcal{H}3$ implication $h \Leftarrow \alpha$ into (a linear number of) $\mathcal{H}3$-special implications, we consider the dependence graph D_α of the pre-condition α. As this graph is acyclic, we can *direct* its edges in such a way that the resulting graph D'_α has the following properties:

- D'_α is a forest with edges directed towards the roots;
- every head variable only occurs in roots.

Such an edge orientation can be determined by a DFS traversal of the graph D_α. The occurrence of a head variable in a root is considered as the *definition* of potential bindings of the variable. According to the property H3, each variable X which occurs more than once in α, corresponds to a unique edge (g_1, g_2) in D'_α from g_1 to g_2. The occurrence of X in the literal g_2 is the *use* of X whereas the occurrence of X in the literal g_1 is the *definition* of potential bindings of X.

Example 4. Consider the following implication:

$$p(a(X,Y)) \quad \Leftarrow \quad q_1(d(Z,c(X))), q_2(Y), q_3(Z), s(b,b)$$

The corresponding directed graph D'_α is shown in fig. 1. In this figure, we added

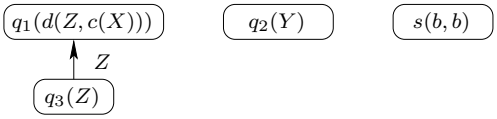


Fig. 1. The directed dependence graph D'_α.

to the single edge the variable by which it is induced. Note that there is one root besides those containing head variables, namely the literal $s(b,b)$. □

Assuming that the variables of distinct implications all are distinct, the preprocessing transformation proceeds as follows:

- For every subterm t of c, we introduce a new predicate h_t. If $t \equiv f(t_1,\ldots,t_k)$, the predicate h_t is defined by the implication:

$$h_t(f(Z_1,\ldots,Z_k)) \quad \Leftarrow \quad h_{t_1}(Z_1),\ldots,h_{t_k}(Z_k)$$

- For every root $g \equiv q(t_1,\ldots,t_k)$ of a pre-condition which does not contain a head variable, we introduce a new nullary predicate h'_g which is defined by the implication:

$$h'_g() \quad \Leftarrow \quad q(X_1,\ldots,X_k), h_{t_1}(X_1),\ldots,h_{t_k}(X_k)$$

- For an implication $p(t_1,\ldots,t_k) \Leftarrow \alpha$, we introduce the implication:

$$p(Z_1,\ldots,Z_k) \quad \Leftarrow \quad h_{t_1}(Z_1),\ldots,h_{t_k}(Z_k), \mathbf{q}$$

  where $\mathbf{q}$ is the sequence of queries $h'_g()$, g a query of α without head variable.
- It remains to define the predicates h_X. If X occurs only once, we set:

$$h_X(Z) \quad \Leftarrow \quad h_\top(Z)$$

  where $h_\top$ is a unary predicate which holds for all trees t. Since the set of constructors is fixed, $h_\top$ can be defined by a normal clause of size $\mathcal{O}(1)$. Every other variable X has a defining occurrence in some query g in some pre-condition. If $g \equiv q(X)$, then we add the implication:

$$h_X(Z) \quad \Leftarrow \quad q(Z)$$

  Otherwise, if $g \equiv q(s_1,\ldots,s_m)$, we introduce new unary predicates g_s for every super-term s of X occurring in g together with the implications:

$$\begin{array}{lll}
h_X(Z) & \Leftarrow & g_X(Z) \\
g_{s_i}(Z_i) & \Leftarrow & q(Z_1,\ldots,Z_m), h_{s_1}(Z_1),\ldots,h_{s_{i-1}}(Z_{i-1}), h_{s_{i+1}}(Z_{i+1}),\ldots,h_{s_m}(Z_m) \\
 & & \text{if } X \text{ occurs in } s_i \\
g_{s'_j}(Z_j) & \Leftarrow & g_{s'}(f(Z_1,\ldots,Z_{m'})), h_{s'_1}(Z_1),\ldots,h_{s'_{j-1}}(Z_{j-1}), h_{s'_{j+1}}(Z_{j+1}),\ldots,h_{s'_{m'}}(Z_{m'}) \\
 & & \text{if } s' \equiv f(s'_1,\ldots,s'_{m'}) \text{ occurs in } g \text{ and } X \text{ occurs in } s'_j
\end{array}$$

Example 5. Applying these rules to the implication from example 4, we obtain:

$$
\begin{array}{lcll}
h_{a(X,Y)}(a(X',Y')) & \Leftarrow & h_X(X'), h_Y(Y') & h_Y(Y') \quad \Leftarrow \quad q_2(Y') \\
h_X(X') & \Leftarrow & g_X(X') & h_Z(Z') \quad \Leftarrow \quad q_3(Z') \\
h_b(b) & \Leftarrow & & \\
g_{c(X)}(X') & \Leftarrow & g_{d(Z,c(X))}(d(Z',X')), h_Z(Z') & \\
g_X(X') & \Leftarrow & g_{c(X)}(c(X')) & g_{d(Z,c(X))}(X') \quad \Leftarrow \quad q_1(X') \\
h'_{s(b,b)}() & \Leftarrow & s(X_1,X_2), h_b(X_1), h_b(X_2) & \\
p(X') & \Leftarrow & h_{a(X,Y)}(X'), h'_{s(b,b)}() &
\end{array}
$$

where g denotes the query $\quad q_1(d(Z, c(X)))$. □

Indeed, this transformation introduces $\mathcal{H}3$-special implications only. Note also that every query may contain the defining occurrence of at most one variable. Therefore, the transformed clause is of size $\mathcal{O}(|c|)$. By fixpoint induction, it can be proven that the transformed clause is equivalent to c. Since the transformation can be implemented in linear time, the statement of proposition 3 follows. □

Given a linear time pre-processing phase, it therefore suffices to construct in time $\mathcal{O}(|c|^3)$, for every $\mathcal{H}3$-special Horn clause c, an equivalent normal clause of size $\mathcal{O}(|c|^2)$. Here, we only collect the necessary normalization rules.

Assume that $\mathcal{H}$ is the least model of the subset of normal implications in c. If an implication equals $p(X_1) \Leftarrow q_1(X_1)$, then for every normal implication:

$$q_1(f(X_1,\ldots,X_k)) \quad \Leftarrow \quad p_1(X_1),\ldots,p_k(X_k)$$

we add the implication:

$$p(f(X_1,\ldots,X_k)) \quad \Leftarrow \quad p_1(X_1),\ldots,p_k(X_k)$$

Now consider non-normal implications of the forms (1) through (4).
Case 1: The implication equals $\quad p(X_1,\ldots,X_k) \quad \Leftarrow \quad q_1(X_1),\ldots,q_k(X_k),\mathbf{q}$ where $\quad \mathbf{q} \equiv f_1(),\ldots,f_m()$ is a non-empty sequence of nullary queries. If $\mathcal{H}(f_j) \not\models \emptyset$, for every j, then we add:

$$p(X_1,\ldots,X_k) \Leftarrow q_1(X_1),\ldots,q_k(X_k)$$

Case 2: The implication equals $\quad p(X_i) \Leftarrow q(f(X_1,\ldots,X_k)),\mathbf{p}$ for a sequence

$$\mathbf{p} \quad \equiv \quad p_1(X_1),\ldots,p_{i-1}(X_{i-1}),p_{i+1}(X_{i+1}),\ldots,p_k(X_k)$$

Then for every normal implication:

$$q(f(X_1,\ldots,X_k)) \quad \Leftarrow \quad q_1(X_1),\ldots,q_k(X_k)$$

we add: $\quad p(X_i) \Leftarrow q_i(X_i)$ — provided that $\mathcal{H}(p_j) \cap \mathcal{H}(q_j) \not\models \emptyset$ for all $j \not\models i$.
Case 3: The implication equals $\quad p(X_i) \Leftarrow q(X_1,\ldots,X_k),\mathbf{p}$ for $k > 1$ and

$$\mathbf{p} \quad \equiv \quad p_1(X_1),\ldots,p_{i-1}(X_{i-1}),p_{i+1}(X_{i+1}),\ldots,p_k(X_k)$$

Then we proceed similar to *Case 2*. Thus, for every normal implication:

$$q(X_1,\ldots,X_k) \quad \Leftarrow \quad q_1(X_1),\ldots,q_k(X_k)$$

we add: $p(X_i) \Leftarrow q_i(X_i)$ — provided that $\mathcal{H}(p_j) \cap \mathcal{H}(q_j) \neq \emptyset$ for all $j \neq i$.
Case 4: The implication equals $p() \Leftarrow q(X_1, \ldots, X_k), p_1(X_1), \ldots, p_k(X_k)$. Then we add: $p() \Leftarrow \epsilon$ — provided that either $k = 1$ and $\mathcal{H}(q)$ and $\mathcal{H}(p_1)$ have a non-empty intersection or $k > 1$ and there is a normal implication:

$$q(X_1, \ldots, X_k) \quad \Leftarrow \quad q_1(X_1), \ldots, q_k(X_k)$$

such that $\mathcal{H}(q_i) \cap \mathcal{H}(p_i) \neq \emptyset$ for all i.

At most $|c|^2$ constraints of the form $p(X) \Leftarrow q(X)$ can be generated. Only $\mathcal{O}(|c|)$ normal constraints are generated for each predicate. So, the total number of generated constraints is $\mathcal{O}(|c|^2)$. Each original constraint as well as as each generated constraint of type (1) triggers at most $|c|$ normalization steps, which altogether can be implemented in time $\mathcal{O}(|c|^3)$ — given that non-emptiness of all (occurring) pairs of unary predicates can be decided in cubic time. The trivial upper bound for these tests is $\mathcal{O}(|c|^4)$. A more detailed analysis, however, reveals that time $\mathcal{O}(|c|^3)$ suffices. Thus, normalization of $\mathcal{H}3$ clauses is cubic. □

6 Application

We have seen that some implications naturally arising in the formalization of the Spi analysis, do not have property H3. It turns out that some further implications even violate H1 (see appendix A for a complete specification of the analysis).

Example 6. The evaluation rule of *hasValue* for the constructor enc is given by:

$$\begin{array}{lcl} \mathit{hasValue}(\mathsf{enc}(E,K), \mathsf{enc}(V,KV)) & \Leftarrow & \mathit{occurs}(\mathsf{enc}(E,K)), \\ & & \mathit{hasValue}(E,V), \\ & & \mathit{hasValue}(K,KV) \end{array}$$

This implication does not comply with H1, since the head variables E, K and V, KV are connected without being siblings. □

Each complicated implication of the Spi analysis, however, contains either a query to one of the predicates *reach* or *occurs*. A closer inspection reveals that both predicates hold only for *subterms* of the ground term prog. Therefore, we obtain an equivalent clause if we instantiate the variables in these queries in all ways such that the arguments of *reach* and *occurs* are subterms of prog.

Example 7. Consider the implications for *hasValue* in the examples 3 and 6. Instantiating the argument positions of *reach* and *occurs* results in the clauses:

$$\begin{array}{lcl} \mathit{hasValue}(x,Y) & \Leftarrow & \mathit{reach}(\mathsf{decrypt}(e,x,k,t)), \\ & & \mathit{hasValue}(k,V), \\ & & \mathit{hasValue}(e,\mathsf{enc}(Y,V)) \\ \mathit{hasValue}(\mathsf{enc}(e,k), \mathsf{enc}(V,VK)) & \Leftarrow & \mathit{occurs}(\mathsf{enc}(e,k)), \\ & & \mathit{hasValue}(e,V), \\ & & \mathit{hasValue}(k,KV) \end{array}$$

for all sub-programs $t' = \mathsf{decrypt}(e,x,k,t)$ and occurring expressions $\mathsf{enc}(y,k)$. Here, x, e, k and t are ground terms. Thus, the resulting implications are H3. □

This instantiation of the Spi analysis indeed results in H3 implications only. Since we use a succinct representation, multiple occurrences of the same ground subterm are represented only once. Therefore, the size of the clause is increased by a constant factor only. We conclude:

Theorem 4. *Control-flow analysis for Spi is cubic.* □

7 Conclusion

We presented a new class $\mathcal{H}1$ of Horn clauses whose least model can be computed exactly, and exhibited subclasses $\mathcal{H}2$ and $\mathcal{H}3$ where this can be done in polynomial time. Due to its expressivity, we find the class $\mathcal{H}1$ interesting in its own right. The most practical class, though, seems to be the class $\mathcal{H}3$, since it admits normalization in cubic time. We exemplified this by applying the normalization procedure for $\mathcal{H}3$ to construct a cubic time analysis for Spi. Since the "natural" formulation of the analysis did not meet the syntactical requirements of $\mathcal{H}3$, we massaged the original formulation by instantiating some parameters in the clause with finitely many ground subterms. The underlying idea can be generalized. It remains for future work to systematically explore corresponding clause transformations w.r.t. their strengths and complexities.

References

1. M. Abadi and A.D. Gordon. A Calculus for Cryptographic Protocols - The Spi Calculus. *Information and Computation*, 148:1–70, January 1999.
2. A. Aiken. Introduction to Set Constraint-Based Program Analysis. *Science of Computer Programming (SCP)*, 35(2):79–111, 1999.
3. D.A. Basin and H. Ganzinger. Complexity Analysis Based on Ordered Resolution. *Journal of the ACM*, 48(1):70–109, 2001.
4. W. Charatonik and A. Podelski. Set Constraints with Intersection. In *12th Ann. IEEE Symp. on Logic in Computer Science (LICS)*, 362–372, 1997.
5. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1999.
6. T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic Programs as Types of Logic Programs. In *6th Symp. on Logic in Computer Science (LICS)*, 300–309, 1991.
7. H. Ganzinger and D.A. McAllester. A New Meta-complexity Theorem for Bottom-Up Logic Programs. In *First Int. Joint Conference on Automated Reasoning (IJCAR)*, 514–528. LNCS 2083, 2001.
8. N. Heintze and J. Jaffar. A Decision Procedure for a Class of Set Constraints. In *5th Ann. IEEE Symp. on Logic in Computer Science (LICS)*, 42–51, 1990.
9. N.D. Jones and S.S. Muchnick. Complexity of Flow Analysis, Inductive Assertion Synthesis, and a Language due to Dijkstra. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 12, 380–393. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.
10. D. Lugiez and P. Schnoebelen. Decidable First-Order Transition Logics for PA-Processes. In *27th Int. Coll. on Automata, Languages and Programming (ICALP)*, 342–353. LNCS 1853, 2000.
11. D. McAllester. On the Complexity Analysis of Static Analyses. In *6th Static Analysis Symposium (SAS)*, 312–329. LNCS 1694, 1999.

12. F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis.* Springer, 1999.
13. F. Nielson, H. Riis Nielson, and H. Seidl. Automatic Complexity Analysis. In *European Symposium on Programming (ESOP)*, 243–261. LNCS 2305, 2002.
14. F. Nielson, H. Riis Nielson, and H. Seidl. Cryptographic Analysis in Cubic Time. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, volume 62. Elsevier Science Publishers, 2002.
15. F. Nielson and H. Seidl. Control-Flow Analysis in Cubic Time. In *European Symposium on Programming (ESOP)*, 252–268. LNCS 2028, 2001.
16. F. Nielson and H. Seidl. Succinct Solvers. Tech. Report 01-12, Trier, 2001.
17. L. Pacholski and A. Podelski. Set Constraints - a Pearl in Research on Constraints. In Gert Smolka, editor, *3rd Int. Conf. on Principles and Practice of Constraint Programming (CP)*, volume 1330 of *Springer LNCS*, 549–561. Springer-Verlag, 1997.
18. H. Seidl. Haskell Overloading is DEXPTIME Complete. *Information Processing Letters (IPL)*, 54:57–60, 1994.

A The Clauses for the Spi Analysis

For simplicity, we consider only *unary* encryptions and *unary* send and receive operations. Here are once again the implications for decryption:

$$\begin{array}{lcl} hasValue(X,Y), reach(T) & \Leftarrow & reach(\mathsf{decrypt}(E,X,K,T)), \\ & & hasValue(K,V), \\ & & hasValue(E,\mathsf{enc}(Y,V)) \\ occurs(E), occurs(K) & \Leftarrow & reach(\mathsf{decrypt}(E,X,K,T)) \end{array}$$

Similar rules model the remaining syntactical constructs. For send and recv, we obtain:

$$\begin{array}{lcl} hasMessage(CV,V), reach(T) & \Leftarrow & reach(\mathsf{send}(C,E,T)), \\ & & hasValue(C,CV), \\ & & hasValue(E,V) \\ occurs(C), occurs(E) & \Leftarrow & reach(\mathsf{send}(C,E,T)) \\ hasValue(X,V), reach(T) & \Leftarrow & reach(\mathsf{recv}(C,X,T)), \\ & & hasValue(C,CV), \\ & & hasMessage(CV,V) \\ occurs(C) & \Leftarrow & reach(\mathsf{recv}(C,\_,\_)) \end{array}$$

For control constructs, we have:

$$\begin{array}{lcl} reach(T) & \Leftarrow & reach(\mathsf{bang}(T)) \\ reach(T_1), reach(T_2) & \Leftarrow & reach(\mathsf{par}(T_1,T_2)) \\ reach(T_0) & \Leftarrow & reach(\mathsf{case}(E,\_,T_0,\_)), \\ & & hasValue(E,\mathsf{zero}) \\ hasValue(X,V), reach(T_1) & \Leftarrow & reach(\mathsf{case}(E,X,\_,T_1)), \\ & & hasValue(E,\mathsf{succ}(V)) \\ occurs(E) & \Leftarrow & reach(\mathsf{case}(E,\_,\_,\_)) \\ hasValue(X_1,V_1), hasValue(X_2,V_2), reach(T) & \Leftarrow & reach(\mathsf{let}(E,X_1,X_2,T)), \\ & & hasValue(E,\mathsf{pair}(V_1,V_2)) \\ occurs(E) & \Leftarrow & reach(\mathsf{let}(E,\_,\_,\_)) \end{array}$$

Besides encryption and the atom zero, the Spi calculus has two further data constructors, the unary succ and the binary pair. The evaluation rules for these are given by:

$$
\begin{array}{lcl}
occurs(E) & \Leftarrow & occurs(\mathsf{succ}(E)) \\
occurs(E_1), occurs(E_2) & \Leftarrow & occurs(\mathsf{pair}(E_1, E_2)) \\
occurs(E), occurs(K) & \Leftarrow & occurs(\mathsf{enc}(E, K))
\end{array}
$$

$$
\begin{array}{lcl}
hasValue(\mathsf{zero}, \mathsf{zero}) & \Leftarrow & \\
hasValue(\mathsf{succ}(E), \mathsf{succ}(V)) & \Leftarrow & occurs(\mathsf{succ}(E)), \\
 & & hasValue(E, V) \\
hasValue(\mathsf{pair}(E_1, E_2), \mathsf{pair}(V_1, V_2)) & \Leftarrow & occurs(\mathsf{pair}(E_1, E_2)), \\
 & & hasValue(E_1, V_1), \\
 & & hasValue(E_2, V_2) \\
hasValue(\mathsf{enc}(E, K), \mathsf{enc}(V, KV)) & \Leftarrow & occurs(\mathsf{enc}(E, K)), \\
 & & hasValue(E, V), \\
 & & hasValue(K, KV)
\end{array}
$$

Nonuniform Alias Analysis of Recursive Data Structures and Arrays*

Arnaud Venet

LIX, École Polytechnique
91128 Palaiseau France
venet@lix.polytechnique.fr

Abstract. In this paper we present an alias analysis which is able to infer position-dependent equality relationships between pointers in recursively defined data structures or arrays. Our work is based on a semantic model of the execution of a program in which every allocated object is identified by a timestamp which represents the state of the program at the instant of the object creation. We provide a simple numerical abstraction of timestamps which is accurate enough to distinguish between elements of arrays or list-like structures. We follow the methodology of Abstract Interpretation to derive a sound approximation of the program semantics from this abstraction. The computability of our analysis is then guaranteed by using an abstract numerical lattice to represent relations between timestamps.

1 Introduction

Most program manipulation techniques, from compile-time optimization to automatic verification, rely on an alias analysis to ensure the soundness of their results. The precision of the analysis has a substantial impact on the effectiveness of the applications which use the aliasing information [15]. However, among the numerous alias analyses existing in the literature few can provide accurate information in the presence of recursive data structures [10,11,12,21,23,24]. The analyses described in [12] and [21] can infer the shape of a structure (for example a list or a tree) and prove that this shape is preserved under certain operations. Deutsch [10,11] has been the first one to propose a *nonuniform* alias analysis, that is an analysis where the aliasing relationships are parameterized by the position of the elements in recursive data structures. In a previous work [23,24] we have shown how to extend the applicability and precision of Deutsch's analysis by using a richer abstract domain. In this paper we present a new technique for computing position-dependent aliasing information which is both simpler and more expressive than the existing models.

The nonuniform alias analyses designed so far [10,11,23,24] rely on a storeless model of the memory introduced by Jonkers [16] in which aliasing is explicitly

* This work was supported in part by the RTD project IST-1999-20527 DAEDALUS of the european IST FP5 programme.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 36–51, 2002.
© Springer-Verlag Berlin Heidelberg 2002

```
void copy(List l1, List l2) {
  List p1, p2;
  p1 = l1;
  p2 = l2;
  while(p1 != null) {
    p1->val = p2->val;
    p1 = p1->next;
    p2 = p2->next;
  }
}
```

Let c be a counter for the loop, then:

$\texttt{p1}(\texttt{->next})^i\texttt{->val} \equiv \texttt{l1}(\texttt{->next})^j\texttt{->val}$

$\texttt{p2}(\texttt{->next})^i\texttt{->val} \equiv \texttt{l2}(\texttt{->next})^j\texttt{->val}$

for all i, j such that $j = i + c$

As a consequence of these relations:

$\texttt{l1}(\texttt{->next})^i\texttt{->val} \equiv \texttt{l2}(\texttt{->next})^j\texttt{->val}$

for all i, j such that $j = i$

After abstraction we have:

$\texttt{p1}(\texttt{->next})^*\texttt{->val} \equiv \texttt{l1}(\texttt{->next})^*\texttt{->val}$

$\texttt{p2}(\texttt{->next})^*\texttt{->val} \equiv \texttt{l2}(\texttt{->next})^*\texttt{->val}$

$\texttt{l1}(\texttt{->next})^*\texttt{->val} \equiv \texttt{l2}(\texttt{->next})^*\texttt{->val}$

Fig. 1. Limitation of analyses based on an aliasing relation

given by means of an equivalence relation over the access paths into the data structures manipulated by the program. The key idea of the abstraction is to represent the position of an element in a recursive structure by a tuple of counters denoting the number of times each recursive component of the structure has to be unfolded to give access to this element. Nonuniform aliasing relationships can then be expressed by using linear constraints on these counters. For instance, these analyses can discover that the n-th element of the list returned by a list copy function can only be aliased to n-th element of the list given as an argument to the function. However, this model has some drawbacks:

- Representing aliasing explicitly by a binary relation over access paths is costly (the number of alias pairs can increase quadratically by means of transitive closure) and redundant (if two elements in a structure are aliased, then all their common descendants are).
- The aliasing information remains precise as long as recursive structures are manipulated recursively, that is bottom-up. In the case of a list which is modified by an iterative top-down algorithm, these analyses will give poor results as illustrated in Fig. 1 where $\equiv$ denotes the aliasing relation. The loss of precision comes from the fact that these analyses cannot represent relationships between more than two data structures. In the iterative list copy program of the figure, the aliasing relationships between `l1` and `p1` on one hand, and `l2` and `p2` on the other hand are related via an implicit loop counter. Since this information is lost by the abstraction, these analyses will compute a uniform approximation of the aliasing. The problem is that in practice list operations are implemented iteratively for efficiency purposes.
- There is no obvious way of representing array structures in these models since it would imply considering paths of data accessors which may contain integers, i.e. word languages over an infinite signature. It means that all classical algorithms over finite automata which underlie these analyses should be redesigned, which represents a considerable effort. This is a major restric-

tion, since arrays are pervasively used in common programming languages as object containers.

We propose to relieve these limitations by using a different semantic model as a basis for a nonuniform alias analysis which can cope with recursive data structures and arrays as well.

We start with a store-based semantic model in which the memory is described by a graph. Classically, in these models newly allocated objects are anonymously represented by fresh nodes which are added to the store. Instead, we explicitly identify each object with a timestamp which is an abstraction of the execution state at the moment of the object creation. This semantics of memory allocation was suggested in [9]. Burke, Carini, Choi and Hind [3] use a similar idea but they apply a k-limiting scheme which does not allow their analysis to distinguish between elements nested beyond depth k in a structure. In our model we abstract a timestamp by an integer which intuitively represents the number of times the corresponding allocation instruction has been executed in the current function call. This abstraction is precise enough to deal with common algorithms for iterating over a list or an array. We specify our analysis in the framework of Abstract Interpretation [5,6,4,7] by deriving an approximation of the program semantics from an abstract representation of the memory.

In order to construct the abstract domain we need to know the shape of the memory graph. We use Steensgaard's analysis [22] for simplicity, but this information can be provided by any extant pointer analysis. The edges of this graph are then decorated with numerical constraints relating the timestamps of the source and target nodes together with the element index in the case of arrays. These enriched storage graphs form our abstract representation of memory. Therefore, the abstract semantics simply amounts to performing arithmetic operations on integers. Array operations fit smoothly in this model and the analysis can infer nonuniform aliasing information relating arrays and recursive structures by means of numerical relations between timestamps. As far as we know, this is the first static analysis which can provide position-dependent aliasing information in the presence of arrays.

The paper is organized as follows. In Sect. 2 we define the syntax and semantics of the simple first-order programming language that we will use to describe our analysis. Section 3 presents the abstract interpretation framework which underlies the design of the analysis. In Sect. 4 we show how to construct the abstract domain upon the abstract storage graph provided by a prior static analysis. We sketch how to compute this abstract graph by using Steensgaard's algorithm. Section 5 describes the abstract semantics of a program. We discuss extensions to this analysis in Sect. 6.

2 Definition of the Language

We will describe our analysis by using a simply-typed first-order imperative language. There are only two types of values in our language: integers, denoted by the type ι, and pointers, denoted by the type o. We assume that we are

provided with a set $\mathcal{V}$ of variable names and a set $\mathcal{F}$ of symbols denoting the fields of structured objects. A program P is a set of function declarations:

$$\textbf{fun } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = \textbf{local } v_1 : \tau'_1, \ldots, v_n : \tau'_k \textbf{ in } cmd$$

where the τ_i, τ'_j's and τ are types, the x_i's represent the function parameters and the v_i's are local variables. A command *cmd* is defined by the following grammar:

cmd ::=	$\{cmd\ ;\ \ldots\ ;\ cmd\}$	sequential composition
|	**while** *bool* **do** *cmd*	iteration
|	**if** *bool* **then** *cmd* **else** *cmd*	condition
|	**return** x	function return
|	$x := f(x_1, \ldots, x_n)$	function call
|	$x :=$ **malloc**	memory allocation
|	$x[i] := y$	array update
|	$x.f := y$	field update
|	$x := y[i]$	array access
|	$x := y.f$	field access
|	$x := expr$	simple assignment

where $x, y, i, x_1, \ldots, x_n \in \mathcal{V}$ and $f \in \mathcal{F}$. The syntax of expressions is defined as follows:

$expr$ ::=	**null**	null reference
|	n	integer constant
|	$x + n$	incrementation
$bool$ ::=	$x < y \mid x < n \mid x =$ **null**	boolean expressions
|	*bool* & *bool* | **not** *bool*	

where $x, y \in \mathcal{V}$ and $n \in \mathbb{N}$. We assume that the variables appearing respectively on the lefthand and righthand sides of an assignment are distinct. The syntax is intentionally minimalist for the sake of clarity. We assume that programs are well-formed with respect to the usual context rules and well-typed with respect to types ι and o. We use a very loose representation of memory: an object does not hold any information about its type or size. All memory operations are valid except when a null pointer is involved. We represent the store by a graph in which the nodes are the allocated objects and the edges denote fields of structures or elements of arrays. Therefore, in this model an object can be both viewed as an array and a structure, and both kinds of operations apply. The purpose is to simplify the definition of the semantic rules.

The execution of a program P is modelled by a small-step operational semantics. We suppose that every command *cmd* in the program is uniquely labelled by $\ell : cmd$. We denote by $\mathcal{L}$ the set of labels in P. For each $\ell : cmd$ whre *cmd* is not a **return** command, we denote by $\text{next}(\ell)$ the label of the unique command which follows *cmd* in the control flow graph of the enclosing function. We denote by $\text{entry}(f)$ the label of the entry point of a function f, by $\text{params}(f)$ the tuple of formal parameters of f and by $\text{locals}(f)$ the set of local variables of f. We

$$\frac{}{\langle S, \ell : \{\ell' : cmd\ ;\ \ldots\}, \varrho, M\rangle \rightarrow \langle S, \ell' : cmd, \varrho, M\rangle}$$

$$\frac{[\![b]\!]\varrho = \mathbf{true},\ \varrho' = \varrho\{\mathbf{c}_\ell \leftarrow \varrho(\mathbf{c}_\ell) + 1\}}{\langle S, \ell : \mathbf{while}\ b\ \mathbf{do}\ \ell' : cmd, \varrho, M\rangle \rightarrow \langle S, \ell' : cmd, \varrho', M\rangle}$$

$$\frac{[\![b]\!]\varrho = \mathbf{false}}{\langle S, \ell : \mathbf{while}\ b\ \mathbf{do}\ \ell' : cmd, \varrho, M\rangle \rightarrow \langle S, \mathrm{next}(\ell), \varrho, M\rangle}$$

$$\frac{[\![b]\!]\varrho = \mathbf{true}}{\langle S, \ell : \mathbf{if}\ b\ \mathbf{then}\ \ell_1 : cmd_1\ \mathbf{else}\ \ell_2 : cmd_2, \varrho, M\rangle \rightarrow \langle S, \ell_1 : cmd_1, \varrho, M\rangle}$$

$$\frac{[\![b]\!]\varrho = \mathbf{false}}{\langle S, \ell : \mathbf{if}\ b\ \mathbf{then}\ \ell_1 : cmd_1\ \mathbf{else}\ \ell_2 : cmd_2, \varrho, M\rangle \rightarrow \langle S, \ell_2 : cmd_2, \varrho, M\rangle}$$

$$\frac{S = S'.(\ell' : y := f(x_1, \ldots, x_n), \varrho')}{\langle S, \ell : \mathbf{return}\ x, \varrho, M\rangle \rightarrow \langle S', \mathrm{next}(\ell'), \varrho'\{y \leftarrow \varrho(x)\}, M\rangle}$$

$$\frac{\begin{array}{c}\text{for all } x \in \mathrm{locals}(f) : \varrho'(x) = \begin{cases} \bot_{\mathcal{T}} & \text{if } x : o \\ 0 & \text{if } x : \iota \end{cases}, \mathrm{params}(f) = (p_1, \ldots, p_n), \\ \text{for all } 1 \leq i \leq n : \varrho'(p_i) = \varrho(x_i), \text{for all } \ell \in \mathrm{loops}(f) : \varrho'(\mathbf{c}_\ell) = 0 \end{array}}{\langle S, \ell : y := f(x_1, \ldots, x_n), \varrho, M\rangle \rightarrow \langle S.(\ell, \varrho), \mathrm{entry}(f), \varrho', M\rangle}$$

Fig. 2. Concrete semantics of the language: control structures.

choose a model of the memory suggested by Deutsch [9] where all objects are uniquely identified with a timestamp which is an image of the execution state at the instant of the object creation. The sequence of return points occurring in the call stack at the execution of a **malloc** seems the most obvious choice for timestamps. However, this information is not sufficient to identify uniquely an object because of loops. The idea is to enrich this timestamp by attaching to each return point in the call stack the number of times every loop of the enclosing function has been executed. Then we can unambiguously refer to an object via its timestamp.

More formally, we denote by $\mathrm{loops}(f)$ the set of labels of all **while** loops occurring in the function f. We associate to each $\ell \in \mathrm{loops}(f)$ a distinct counter $\mathbf{c}_\ell$ in $\mathcal{V}$. A timestamp is a sequence $(\ell_1, \varrho_1) \ldots (\ell_n, \varrho_n)$ where, for each $1 \leq i \leq n$, ℓ_i is the label of a function call, and ϱ_i is an environment mapping every loop counter of the function containing ℓ_i to an integer. We denote by $\bot_{\mathcal{T}}$ the special timestamp representing a null pointer. Let $\mathcal{T}$ be the set of all timestamps. We denote by $\mathrm{vars}(f)$ the set of local variables, formal parameters and loop counters of the function f. A state s of the operational semantics is a tuple $\langle S, \ell, \varrho, M\rangle$,

$$\frac{\sigma = \mathrm{stamp}(S.(\ell, \varrho))}{\langle S, \ell : x := \textbf{malloc}, \varrho, (N, E)\rangle \to \langle S, \mathrm{next}(\ell), \varrho\{x \leftarrow \sigma\}, (N \cup \{\sigma\}, E)\rangle}$$

$$\frac{\varrho(x) = \bot_{\mathcal{T}}}{\langle S, \ell : \begin{cases} x.f := y \\ x[i] := y \end{cases}, \varrho, M\rangle \to \Omega}$$

$$\frac{\varrho(x) \neq \bot_{\mathcal{T}}, E' = E - (\{\varrho(x)\} \times \{f\} \times \mathcal{S})}{\langle S, \ell : x.f := y, \varrho, (N, E)\rangle \to \langle S, \mathrm{next}(\ell), \varrho, (N, E' \cup \{(\varrho(x), f, \varrho(y))\})\rangle}$$

$$\frac{\varrho(x) \neq \bot_{\mathcal{T}}, E' = E - (\{\varrho(x)\} \times \{\varrho(i)\} \times \mathcal{T})}{\langle S, \ell : x[i] := y, \varrho, (N, E)\rangle \to \langle S, \mathrm{next}(\ell), \varrho, (N, E' \cup \{(\varrho(x), \varrho(i), \varrho(y))\})\rangle}$$

$$\frac{\varrho(y) = \bot_{\mathcal{T}}}{\langle S, \ell : \begin{cases} x := y.f \\ x := y[i] \end{cases}, \varrho, M\rangle \to \Omega}$$

$$\frac{\varrho(y) \neq \bot_{\mathcal{T}}, \sigma = \begin{cases} \sigma' & \text{if } (\varrho(y), f, \sigma') \in E \\ \bot_{\mathcal{T}} & \text{otherwise} \end{cases}}{\langle S, \ell : x := y.f, \varrho, (N, E)\rangle \to \langle S, \mathrm{next}(\ell), \varrho\{x \leftarrow \sigma\}, (N, E)\rangle}$$

$$\frac{\varrho(y) \neq \bot_{\mathcal{T}}, \sigma = \begin{cases} \sigma' & \text{if } (\varrho(y), \varrho(i), \sigma') \in E \\ \bot_{\mathcal{T}} & \text{otherwise} \end{cases}}{\langle S, \ell : x := y[i], \varrho, (N, E)\rangle \to \langle S, \mathrm{next}(\ell), \varrho\{x \leftarrow \sigma\}, (N, E)\rangle}$$

$$\frac{}{\langle S, \ell : x := e, \varrho, M\rangle \to \langle S, \mathrm{next}(\ell), \varrho\{x \leftarrow [\![e]\!]\varrho\}, M\rangle}$$

Fig. 3. Concrete semantics of the language: assignment commands.

where ℓ is the label of a command, ϱ is an environment mapping each $x \in \mathrm{vars}(f)$, where f is the function enclosing the command labelled by ℓ, to a timestamp in $\mathcal{T}$ if x has type o, to an integer otherwise. The component S of the state s is a call stack, that is a sequence $(\ell_1, \varrho_1) \ldots (\ell_n, \varrho_n)$ where, for all $1 \leq i \leq n$, ℓ_i is the label of a function call and ϱ_i is an environment at that point. The component M of s is a memory configuration, that is a labelled directed graph (V, E), where $V \subseteq \mathcal{T}$ and $E \subseteq V \times (\mathcal{F} \cup \mathbb{N}) \times V$. We denote by $\mathcal{M}$ the set of all memory configurations. We also consider a special error state Ω which is used to model runtime errors. We denote by Σ the set of all states s. The operational semantics is then given by a transition relation $\to \in \wp(\Sigma \times \Sigma)$.

If ϱ is an environment, $A \subseteq \mathcal{V}$, $x \in \mathcal{V}$ and $v \in \mathcal{T} \uplus \mathbb{N}$, we denote by $\varrho|_A$ the restriction of the domain of ϱ to the variables of A, and by $\varrho\{x \leftarrow v\}$ the environ-

ment which maps x to v and every variable $y \neq x$ to $\varrho(y)$. If $S = (\ell_1, \varrho_1)...(\ell_n, \varrho_n)$ is a call stack and, for all $1 \leq i \leq n$, f_i is the function enclosing the command labelled by ℓ_i, we define the timestamp corresponding to S as follows:

$$\text{stamp}(S) = (\ell_1, \varrho_1|_{\text{loops}(f_1)}) \ldots (\ell_n, \varrho_n|_{\text{loops}(f_n)})$$

The transition relation of the small-step operational semantics is then defined by a set of rules in Fig. 2 and Fig. 3. Notice that whenever a function is called, all local variables are initialized to 0 or $\bot_{\mathcal{T}}$ accordingly to their type. The semantics $[\![e]\!]$ of an expression maps an environment ϱ to a value of the corresponding type as follows, boolean expressions being interpreted over the usual boolean algebra $(\mathbf{true}, \mathbf{false}, \wedge, \vee, \neg)$:

$$\begin{array}{rcl} [\![\mathbf{null}]\!]\varrho = \bot_{\mathcal{T}} & \qquad & [\![x < y]\!]\varrho = (\varrho(x) < \varrho(y)) \\ [\![n]\!]\varrho = n & & [\![x < n]\!]\varrho = (\varrho(x) < n) \\ [\![x + n]\!]\varrho = \varrho(x) + n & & [\![e_1 \,\&\, e_2]\!]\varrho = [\![e_1]\!]\varrho \wedge [\![e_2]\!]\varrho \\ [\![x = \mathbf{null}]\!]\varrho = (\varrho(x) = \bot_{\mathcal{T}}) & & [\![\mathbf{not}\ e]\!]\varrho = \neg[\![e]\!]\varrho \end{array}$$

We assume that there is an entry function *main* in the program P which takes no arguments. The initial state i_P of the program is then defined as follows:

$$i_P = \langle \varepsilon, \text{entry}(main), \varrho_i, (\emptyset, \emptyset) \rangle$$

where ϱ_i maps every local variable of *main* to 0 or $\bot_{\mathcal{T}}$ accordingly to its type. Our purpose is to analyze the *collecting semantics* $\mathcal{S}$ of P [4] which consists of all the states reachable from i_P by the transition relation:

$$\mathcal{S} = \{s \in \Sigma \mid i_P \xrightarrow{*} s\}$$

Example 1. We consider the following program:

```
make(n : ι) : o =
  local i : ι, cell : o,
      p : o, val : o in {
l1:   while n > 0 do {
l2:     cell := malloc ;
l3:     val := malloc ;
l4:     cell.next := p ;
l5:     cell.value := val ;
l6:     p := cell ;
l7:     i := n - 1 ;
l8:     n := i
      } ;
l9:   return p
}
```

```
copy(l : o) : o =
  local i : ι, j : ι,
        t : o, m : o {
l10:  t := malloc ;
l11:  while not(l = null) do {
l12:    t[i] := l.val ;
l13:    m := l.next ;
l14:    l := m ;
l15:    j := i + 1 ;
l16:    i := j ;
      } ;
l17:  return t
}
```

```
main() : ι =
  local l : o, t : o in {
l18:    l := make (10) ;
l19:    t := copy(l) ;
l20:    return 0
}
```

The function `make` creates a list in a bottom-up way and the function `copy` stores pointers to the elements in an array by a top-down traversal of the list. At program point `l20` we have:

$$\varrho(\mathtt{l}) = (\mathtt{l18}, \{\}).(\mathtt{l2}, \{\mathbf{c}_{11} \mapsto \cancel{1}\})$$
$$\varrho(\mathtt{t}) = (\mathtt{l19}, \{\}).(\mathtt{l10}, \{\mathbf{c}_{111} \mapsto \cancel{0}\})$$

The aliasing between the list and the array are expressed by the following edges in the memory graph:

$$((\mathtt{l18}, \{\}).(\mathtt{l2}, \{\mathbf{c}_{11} \mapsto \cancel{i}\}), \mathtt{value}, (\mathtt{l18}, \{\}).(\mathtt{l3}, \{\mathbf{c}_{11} \mapsto \cancel{i}\}))$$
$$((\mathtt{l18}, \{\}).(\mathtt{l2}, \{\mathbf{c}_{11} \mapsto \cancel{j} + 1\}), \mathtt{next}, (\mathtt{l18}, \{\}).(\mathtt{l2}, \{\mathbf{c}_{11} \mapsto \cancel{j}\}))$$
$$((\mathtt{l18}, \{\}).(\mathtt{l2}, \{\mathbf{c}_{11} \mapsto \cancel{1}\}), \mathtt{next}, \bot_{\mathcal{T}})$$
$$((\mathtt{l19}, \{\}).(\mathtt{l10}, \{\mathbf{c}_{111} \mapsto \cancel{0}\}), i, (\mathtt{l18}, \{\}).(\mathtt{l3}, \{\mathbf{c}_{11} \mapsto \cancel{i}\}))$$

for $1 \leq i \leq 10$, $1 \leq j \leq 9$. □

We use the methodology of Abstract Interpretation to build a sound approximation of the collecting semantics.

3 Abstract Interpretation

The collecting semantics can be expressed as the least fixpoint of the the complete $\cup$-morphism $\mathbb{F}$ defined over the lattice $(\wp(\Sigma), \emptyset, \cup, \Sigma, \cap)$ as follows [4]:

$$\forall X \in \wp(\Sigma) : \mathbb{F}(X) = \{i_P\} \cup \{s \in \Sigma \mid \exists s' \in X : s' \to s\}$$

In general the iterative computation of this fixpoint does not terminate and $\mathcal{S}$ is not finitely representable.

Abstract Interpretation [5,6,4,7] offers various constructive frameworks for computing a safe approximation of $\mathcal{S}$. In this paper we follow the methodology defined in [7] which consists of designing an *abstract semantic specification* $(D^\sharp, \sqsubseteq, \bot^\sharp, \gamma, \mathbb{F}^\sharp, \sqcup, \nabla)$ where $(D^\sharp, \sqsubseteq)$ is a partially ordered set, the *abstract domain*, related to concrete states by a monotonic operator $\gamma : (D^\sharp, \sqsubseteq) \to (\wp(\Sigma), \subseteq)$, the *concretization function*. The abstract semantics is described by a function $\mathbb{F}^\sharp : D^\sharp \to D^\sharp$, satisfying the following soundness condition:

$$\mathbb{F} \circ \gamma \subseteq \gamma \circ \mathbb{F}^\sharp$$

The element $\bot^\sharp$ of $D^\sharp$ provides us with an abstraction of the initial state, that is $\{i_P\} \subseteq \gamma(\bot^\sharp)$. The *join* $\sqcup : D^\sharp \times D^\sharp \to D^\sharp$ is an associative and commutative operation which computes an upper-bound of two elements in $D^\sharp$, and the *widening* $\nabla : D^\sharp \times D^\sharp \to D^\sharp$ is a binary operation over $D^\sharp$ which satisfies the following conditions:

1. $\forall x, y \in D^\sharp : x \sqsubseteq x \nabla y,\ y \sqsubseteq x \nabla y$
2. For every increasing sequence $(x_n)_{x \geq 0}$ of elements of $D^\sharp$, the sequence $(y_n)_{x \geq 0}$ defined as follows:

$$\begin{cases} y_0 & = x_0 \\ y_{n+1} & = y_n \nabla x_{n+1} \end{cases}$$

is ultimately stationary.

Intuitively ∇ can be seen as an upper-bound operation which entails convergence when applied repeatedly.

We use this abstract semantic specification to compute the *abstract iteration sequence* $(\mathbb{F}^\sharp_n)_{n \geq 0}$ which mimicks the iterative fixpoint computation of $\mathbb{F}$:

$$\begin{cases} \mathbb{F}^\sharp_0 & = \bot^\sharp \\ \mathbb{F}^\sharp_{n+1} & = \mathbb{F}^\sharp_n & \text{if } \mathbb{F}^\sharp(\mathbb{F}^\sharp_n) \sqsubseteq \mathbb{F}^\sharp_n \\ & \quad \mathbb{F}^\sharp_n \nabla \mathbb{F}^\sharp(\mathbb{F}^\sharp_n) & \text{otherwise} \end{cases}$$

Theorem 1 (Cousot & Cousot[7]). *The sequence* $(\mathbb{F}^\sharp_n)_{n \geq 0}$ *is ultimately stationary. If* $N \in \mathbb{N}$ *is such that* $\mathbb{F}^\sharp_{N+1} = \mathbb{F}^\sharp_N$*, then for all* $n \geq N$*,* $\mathbb{F}^\sharp_n = \mathbb{F}^\sharp_N$ *and* $\mathcal{S} \subseteq \gamma(\mathbb{F}^\sharp_N)$*.*

This theorem provides us with an effective algorithm for computing an approximation of the collecting semantics $\mathcal{S}$. We will apply this scheme to our alias analysis. First, we construct the abstract domain specification, that is the tuple $(D^\sharp, \sqsubseteq, \bot^\sharp, \gamma, \sqcup, \nabla)$.

4 Construction of the Abstract Domain Specification

The whole abstract specification will rely on the approximation of timestamps. We choose a simple abstraction which only retains the total number of iterations in the topmost environment of a timestamp. More formally, let $\sigma = (\ell_1,_e nv_1) \ldots (\ell_n, \varrho_n) \in \mathcal{T}$ be a timestamp. Let f be the function enclosing the command labelled by ℓ_n. The abstraction $\alpha_\mathcal{T}(\sigma)$ of σ is then defined as follows:

$$\alpha_\mathcal{T}(\sigma) = \sum_{\ell \in \text{loops}(f)} \varrho_n(\mathbf{c}_\ell)$$

The null pointer reference is abstracted by $\alpha_\mathcal{T}(\bot_\mathcal{T}) = 0$. This abstraction loses all information about the shape of the stack and the iteration counters in the callers. However, it is precise enough to capture common iterative object manipulations which are performed within a same function. Moreover the associated semantic operations are fairly simple as we will see in the next section. The specification of our analysis can be readily adapted to more complex timestamp abstractions.

The abstraction of the environments and the memory will be based on relations between integers. Therefore we need a computable representation of tuples of integers. Numerous abstract numerical domains have been developed for

this purpose [17,8,13,14,18,19] with various levels of expressiveness. We leave the choice of such a domain as a parameter of our abstract specification. Following [11], we give a characterization of the primitive operations that an abstract numerical domain implements and we construct the semantic specifications upon these operations. Therefore, choosing a particular numerical domain merely amounts to instanciating these basic operations. More precisely, an abstract numerical domain is a collection of abstract domain specifications $(N_V^\sharp, \sqsubseteq_V, \bot_V^\sharp, \gamma_V : N_V^\sharp \to \wp(\mathbb{N}^V), \sqcup_V, \nabla_V)$ indexed by finite sets of variables $V \subseteq \mathcal{V}$. We denote by $\top_V^\sharp$ an element of $N_V^\sharp$ such that $\gamma_V(\top_V^\sharp) = \mathbb{N}^V$. The primitive operations are characterized as follows:

- If S is a system of linear equality constraints over V, the operation $\mathbf{add}\ S : N_V^\sharp \to N_V^\sharp$ satisfies the following condition:

$$\forall \varrho^\sharp \in N_V^\sharp : \gamma_V(\mathbf{add}\ S(\varrho^\sharp)) \supseteq \{\varrho \in \gamma_V(\varrho^\sharp) \mid \varrho \text{ is a solution of } S\}$$

- If $x_1, \ldots, x_n \in V$, the operation $\mathbf{delete}_{x_1,\ldots,x_n} : N_V^\sharp \to N^\sharp$ satisfies the following condition:

$$\begin{aligned}\forall \varrho^\sharp \in N_V^\sharp : \gamma_V(\mathbf{delete}_{x_1,\ldots,x_n}(\varrho^\sharp)) \supseteq \\ \{\varrho \in \mathbb{N}^V : \exists \varrho' \in \gamma_V(\varrho^\sharp) : \varrho'|_{V-\{x_1,\ldots,x_n\}} = \varrho|_{V-\{x_1,\ldots,x_n\}}\}\end{aligned}$$

- If $x_1, \ldots, x_n \in V$, the operation $\mathbf{project}_{x_1,\ldots,x_n} : N_V^\sharp \to N_{\{x_1,\ldots,x_n\}}^\sharp$ satisfies the following condition:

$$\forall \varrho^\sharp \in N_V^\sharp : \gamma_{\{x_1,\ldots,x_n\}}(\mathbf{project}_{x_1,\ldots,x_n}(\varrho^\sharp)) \supseteq \{\varrho|_{\{x_1,\ldots,x_n\}} \mid \varrho \in \gamma_V(\varrho^\sharp)\}$$

- If $\varrho_1^\sharp \in N_V^\sharp$ and $\varrho_2^\sharp \in N_W^\sharp$, then $\mathbf{mix}(\varrho_1^\sharp, \varrho_2^\sharp)$ is an element of $N_{V\cup W}^\sharp$ which satisfies the following condition:

$$\begin{aligned}\forall \varrho_1 \in \gamma_V(\varrho_1^\sharp) : \forall \varrho_2 \in \gamma_W(\varrho_2^\sharp) : \forall \varrho \in \mathbb{N}^{V\cup W} : \\ (\varrho|_V = \varrho_1 \wedge \varrho|_W = \varrho_2) \Rightarrow \varrho \in \gamma_{V\cup W}(\mathbf{mix}(\varrho_1^\sharp, \varrho_2^\sharp))\end{aligned}$$

Example 2. We consider the abstract numerical domain of Karr [17]. In this domain an element $\rho^\sharp$ of $N_V^\sharp$ is given by a system of linear equality constraints over the variables of V. This system is kept in row-echelon normal form by using Gauss algorithm. The operation $\mathbf{add}\ S(\rho^\sharp)$ simply corresponds to adding the equations of S to the system $\rho^\sharp$ and to normalize the resulting system. The operations $\mathbf{delete}_{x_1,\ldots,x_n}$ consists of eliminating all the constraints involving the variables $x_1, \ldots, x_n$. The operation **project** is dual to **delete**, i.e. we keep all the constraints involving the given variables. The operation $\mathbf{mix}(\rho_1^\sharp, \rho_2^\sharp)$ consists of taking the union of the systems $\rho_1^\sharp$ and $\rho_2^\sharp$. Karr's domain satisfies the ascending chain condition, therefore we can use the join operation $\sqcup_V$ as a widening. □

For each label $\ell \in \mathcal{L}$, if f is the function enclosing the command labelled by ℓ, we denote by $\mathrm{vars}@\ell$ the set of variables $\mathrm{vars}(f)$. We suppose that we have an

abstract domain specification for memory configurations $(\mathcal{M}^\sharp, \sqsubseteq_{\mathcal{M}}, \bot^\sharp_{\mathcal{M}}, \gamma_{\mathcal{M}} : \mathcal{M}^\sharp \to \wp(\mathcal{M}), \sqcup_{\mathcal{M}}, \nabla_{\mathcal{M}})$. Then, the abstract domain $D^\sharp$ is defined as follows:

$$D^\sharp = \mathcal{L} \to \left(N^\sharp_{\text{vars@}\ell} \times M^\sharp \right)$$

The order relation $\sqsubseteq$ over $D^\sharp$ is the pointwise extension of the order relations over the abstract numerical domains and $M^\sharp$. The base element $\bot^\sharp$ is given by $\langle \varrho^\bot_{\text{vars@}\ell}, \bot^\sharp_{\mathcal{M}} \rangle_{\ell \in \mathcal{L}}$ where:

$$\varrho^\bot_{\text{vars@}\ell} = \begin{cases} \textbf{add}\ \{v = 0 \mid v \in \text{vars@}\ell\}(\top^\sharp_{\text{vars@}\ell}) & \text{when } \ell = \text{entry}(\mathit{main}) \\ \bot^\sharp_{\text{vars@}\ell} & \text{otherwise.} \end{cases}$$

If $s^\sharp \in D^\sharp$, we denote by $\langle \varrho(s^\sharp@\ell), M(s^\sharp@\ell) \rangle$ the couple $s^\sharp(\ell)$. The concretization function is then defined as follows:

$$\begin{array}{l} \forall s^\sharp \in D^\sharp : \\ \quad \gamma(s^\sharp) = \{\Omega\} \cup \{(S, \ell, \varrho, M) \mid \varrho \in \gamma_{\text{vars@}\ell}(\varrho(s^\sharp@\ell)) \ \wedge \ M \in \gamma_{\mathcal{M}}(M(s^\sharp@\ell))\} \end{array}$$

This corresponds to the usual partitioning of execution states with respect to the control points [4]. Note that for the sake of simplicity we do not try to handle runtime errors precisely. The widening ∇ and the join $\sqcup$ are defined by pointwise application of the corresponding operators over the component domains.

Now it remains to define the abstract domain specification for memory configurations. Intuitively, an abstract memory configuration is a storage graph in which edges carry relations between the timestamps of the source and target nodes. However, in order to construct this domain we need to know the shape of the memory graph. We can compute the shape of the graph and the numerical relations between timestamps simultaneously, by using the technique of *cofibered domains* [23,24] and an associated widening operator. However, this would make the presentation of the analysis quite complicated. For explanatory purposes, we choose a simpler solution which consists of applying a flow-insensitive pointer analysis prior to our analysis which can infer an abstract storage graph describing the shape of the memory and the values of each pointer variable at every control point. We use the graph produced by this preliminary analysis to construct the abstract domain. We suppose that the results of this analysis are provided in the following form:

- A graph $G^\sharp = (V^\sharp, E^\sharp)$ where $V^\sharp$ is a set of nodes and $E^\sharp \subseteq V^\sharp \times (\mathcal{F} \uplus \{\Box\}) \times V^\sharp$, $\Box$ being a special symbol denoting array elements.
- A function μ mapping every couple (ℓ, x) where $x \in \text{vars@}\ell$ has type o, to a vertex in $G^\sharp$.

Moreover, $G^\sharp$ must be deterministic: $\forall (s, l, t), (s', l', t') \in E^\sharp : (s = s' \wedge l = l') \Rightarrow t = t'$. The abstract graph $G^\sharp$ must also satisfy the following soundness conditions:

$$\forall (S, \ell, \varrho, (V, E)) \in \mathcal{S} : \exists \nu : \mathcal{T} \to V^\sharp : \begin{cases} \forall x \in \text{vars@}\ell : (x : o) \Rightarrow \nu(\varrho(x)) = \mu(\ell, x) \\ \forall (\sigma_1, f, \sigma_2) \in E : (\nu(\sigma_1), f, \nu(\sigma_2)) \in E^\sharp \\ \forall (\sigma_1, n, \sigma_2) \in E : (\nu(\sigma_1), \Box, \nu(\sigma_2)) \in E^\sharp \end{cases}$$

We distinguish three variables $\mathbf{s}, \mathbf{t}, \mathbf{i}$ which will respectively denote the abstract timestamp of the source vertex, the abstract timestamp of the target vertex and the index of an array element for every edge of $G^\sharp$. Then, we define the abstract domain $\mathcal{M}^\sharp$ of memory configurations as:

$$\mathcal{M}^\sharp = \left(\prod_{(s,f,t) \in E^\sharp} N^\sharp_{\{\mathbf{s},\mathbf{t}\}} \right) \times \left(\prod_{(s,\Box,t) \in E^\sharp} N^\sharp_{\{\mathbf{s},\mathbf{i},\mathbf{t}\}} \right)$$

The order relation $\sqsubseteq_{\mathcal{M}}$ is the pointwise extension of the corresponding order relations over the abstract numerical domains. The widening $\nabla_{\mathcal{M}}$ and the join $\sqcup_{\mathcal{M}}$ are defined similarly. The abstract memory configuration $\bot_{\mathcal{M}}$ maps every vertex in $E^\sharp$ to the base element $\bot^\sharp$ of the corresponding numerical lattice. The concretization $\gamma_{\mathcal{M}}(M^\sharp)$ of an abstract memory configuration is the set of memory graphs (V, E) such that there exists a mapping $\nu : \mathcal{T} \to V^\sharp$ satisfying the following conditions:

1. $\forall (\sigma_1, f, \sigma_2) \in E : \langle \mathbf{s} \mapsto \alpha_{\mathcal{T}}(\sigma_1), \mathbf{t} \mapsto \alpha_{\mathcal{T}}(\sigma_2) \rangle \in \gamma_{\{\mathbf{s},\mathbf{t}\}}(M^\sharp(\nu(\sigma_1), f, \nu(\sigma_2)))$
2. $\forall (\sigma_1, n, \sigma_2) \in E :$
$$\langle \mathbf{s} \mapsto \alpha_{\mathcal{T}}(\sigma_1), \mathbf{i} \mapsto n, \mathbf{t} \mapsto \alpha_{\mathcal{T}}(\sigma_2) \rangle \in \gamma_{\{\mathbf{s},\mathbf{i},\mathbf{t}\}}(M^\sharp(\nu(\sigma_1), \Box, \nu(\sigma_2)))$$

This completes the specification of the abstract domain. Note that, even though the preliminary pointer analysis is flow-insensitive, our analysis is flow-sensitive. The shape of the abstract storage graph is the same at all control points, but the annotations which relate the timestamps associated to the nodes of this graph do depend on the control flow.

Steensgaard [22] designed a points-to analysis which can compute the graph $G^\sharp$ in almost-linear time. The algorithm consists of assigning a distinct vertex to each variable of the program. The analysis then constructs an equivalence relation over these vertices, the edges of the graph linking the cosets of this equivalence relation. Every statement in the program is analyzed once resulting into the identification of vertices and/or the addition of new edges. The use of a union-find structure to represent the equivalence relation over the vertices makes this algorithm extremely efficient. We required that the abstract storage graph $G^\sharp$ be deterministic since it simplifies the presentation of our analysis. This implies the use of unification-based pointer analyses like Steensgaard's but it is in no way an intrinsic limitation of our model. We could have formulated the construction of our abstract domain by using an inclusion-based pointer analysis like Andersen's [1] as well.

Example 3. The application of Steensgaard's pointer analysis to the program defined in Example 1 produces the following abstract graph:

$$G^\sharp = (\{n_1, n_2, n_3\}, \{(n_1, \texttt{next}, n_1), (n_1, \texttt{value}, n_2), (n_3, \Box, n_2)\})$$

together with the corresponding assignment of nodes to variables:

$$\{\texttt{t} \mapsto n_3, \texttt{l} \mapsto n_1, \texttt{m} \mapsto n_1, \texttt{cell} \mapsto n_1, \texttt{p} \mapsto n_1, \texttt{val} \mapsto n_2\}$$

$\Box$

1. $[\![x := \mathbf{malloc}]\!]^\sharp\langle\varrho^\sharp, M^\sharp\rangle = \langle[\mathbf{delete}_x\,;\mathbf{add}\,\{x = \sum_{\ell\in\mathrm{loops}(f)}\,\mathbf{c}_\ell\}](\varrho^\sharp), M^\sharp\rangle$
2. $[\![x[i] := y]\!]^\sharp\langle\varrho^\sharp, M^\sharp\rangle = \langle\varrho^\sharp, M^\sharp\{e^\sharp \leftarrow M^\sharp(e^\sharp) \sqcup_{\{\mathbf{s},\mathbf{i},\mathbf{t}\}} T(\varrho^\sharp)\}\rangle$ where
   - $e^\sharp = (\mu(x), \Box, \mu(y))$
   - $T = [\mathbf{add}\,\{\mathbf{s} = x, \mathbf{i} = i, \mathbf{t} = y\}\,;\mathbf{project}_{\mathbf{s},\mathbf{i},\mathbf{t}}]$
3. $[\![x.f := y]\!]^\sharp\langle\varrho^\sharp, M^\sharp\rangle = \langle\varrho^\sharp, M^\sharp\{e^\sharp \leftarrow M^\sharp(e^\sharp) \sqcup_{\{\mathbf{s},\mathbf{t}\}} T(\varrho^\sharp)\}\rangle$ where
   - $e^\sharp = (\mu(x), f, \mu(y))$
   - $T = [\mathbf{add}\,\{\mathbf{s} = x, \mathbf{t} = y\}\,;\mathbf{project}_{\mathbf{s},\mathbf{t}}]$
4. $[\![\ell : x := y[i]]\!]^\sharp\langle\varrho^\sharp, M^\sharp\rangle = \langle T(M^\sharp(e^\sharp), \varrho^\sharp), M^\sharp\rangle$ where
   - $e^\sharp = (\mu(x), \Box, \mu(y))$
   - $T = [\mathbf{mix}\,;\mathbf{delete}_x\,;\mathbf{add}\,\{\mathbf{s} = y, \mathbf{i} = i, \mathbf{t} = x\}\,;\mathbf{project}_{\mathrm{vars}@\ell}]$
5. $[\![\ell : x := y.f]\!]^\sharp\langle\varrho^\sharp, M^\sharp\rangle = \langle T(M^\sharp(e^\sharp), \varrho^\sharp), M^\sharp\rangle$ where
   - $e^\sharp = (\mu(x), f, \mu(y))$
   - $T = [\mathbf{mix}\,;\mathbf{delete}_x\,;\mathbf{add}\,\{\mathbf{s} = y, \mathbf{t} = x\}\,;\mathbf{project}_{\mathrm{vars}@\ell}]$
6. $[\![x := \mathbf{null}]\!]^\sharp\langle\varrho^\sharp, M^\sharp\rangle = \langle[\mathbf{delete}_x\,;\mathbf{add}\,\{x = 0\}](\varrho^\sharp), M^\sharp\rangle$
7. $[\![x := n]\!]^\sharp\langle\varrho^\sharp, M^\sharp\rangle = \langle[\mathbf{delete}_x\,;\mathbf{add}\,\{x = n\}](\varrho^\sharp), M^\sharp\rangle$
8. $[\![x := y + n]\!]^\sharp\langle\varrho^\sharp, M^\sharp\rangle = \langle[\mathbf{delete}_x\,;\mathbf{add}\,\{x = y + n\}](\varrho^\sharp), M^\sharp\rangle$

Fig. 4. Abstract semantics of basic commands.

5 Abstract Semantics of the Language

It now remains to specify the abstract semantic operator $\mathbb{F}^\sharp : D^\sharp \to D^\sharp$. Let $s^\sharp \in D^\sharp$. Following the specification methodology of partitioned semantic transformers [5,4], we will define $\bar{s}^\sharp = \mathbb{F}^\sharp(s^\sharp)$ by a system of semantic equations over the variables $s@\ell$, $\bar{s}@\ell$ for $\ell \in \mathcal{L}$. We will first define the interprocedural behaviour of the program. If f is a function of P we denote by $\mathrm{callers}(f)$ the set of command labels which correspond to a call to f, and by $\mathrm{returns}(f)$ the set of labels which correspond to a **return** command in f. Since our language is first order this information is statically known. For the clarity of presentation, we will denote by $u\,;v$ the composition $v \circ u$ of two operations u and v.

Let f be a function of P with $\mathrm{params}(f) = \{p_1, \ldots, p_n\}$. Let $\ell : x := f(x_1, \ldots, x_n)$ be an element of $\mathrm{callers}(f)$. First, we bijectively replace every variable v occurring in $\varrho(s^\sharp@\ell)$ with a fresh variable v'. Then, we define the abstract function call operation $\mathbf{call}^\sharp$ as follows:

$$\mathbf{call}^\sharp(s^\sharp@\ell) = \langle bind(\mathbf{project}_{x'_1,\ldots,x'_n}(\varrho(s^\sharp@\ell)), init), M(s^\sharp@\ell)\rangle$$

where

- $bind = [\mathbf{mix}\,;\mathbf{add}\,\{p_1 = x'_1, \ldots, p_n = x'_n\}\,;\mathbf{project}_{\mathrm{vars}(f)}]$
- $init = [\mathbf{add}\,\{v = 0 \mid v \in \mathrm{vars}(f) \wedge v \not\in \mathrm{params}(f)\}](\top^\sharp_{\mathrm{vars}(f)})$

Intuitively, this operation amounts to transferring the numerical relationships between the arguments $x_1, \ldots, x_n$ at the caller level into the callee environment. All other local variables of the callee should be initialized to 0. The bijective renaming is necessary to avoid name clashes in the case of a recursive call. The semantic equation associated to the entry point of f can therefore be written as:

$$\bar{s}^\sharp@\mathrm{entry}(f) = \sqcup_{\mathrm{vars}(f)} \{\mathbf{call}^\sharp(s^\sharp@\ell) \mid \ell \in \mathrm{callers}(f)\}$$

Now let $\ell : x := f(x_1, \ldots, x_n)$ be a function call command and $\ell' : \mathbf{return}\ y$ be a function return command in f. First, we bijectively replace every variable v occurring in $\mathrm{vars}(f)$ with a fresh variable v'. Then, we define the abstract function return operation $\mathbf{return}^\sharp_\ell$ as follows:

$$\mathbf{return}^\sharp_\ell(s^\sharp@\ell') = \langle[\mathbf{delete}_x\ ;\ \mathbf{add}\{x = y'\}\ ;\ \mathbf{project}_{\mathrm{vars}@\ell}](return), M(s^\sharp@\ell')\rangle$$

where $return = \mathbf{mix}(\mathbf{project}_{y'}(\varrho(s^\sharp@\ell')), \varrho(s^\sharp@\ell))$. Intuitively, this corresponds to transferring the information about the return value computed at the callee level back into the caller environment. As previously, the bijective renaming is necessary to avoid name clashes in the case of a recursive call.

The abstract semantics of basic assignment commands is defined in Fig. 4. The semantic equation corresponding to a program label ℓ can then be specified as follows:

$$\begin{aligned}\bar{s}^\sharp@\ell = \sqcup_{\mathrm{vars}@\ell}(&\{[\![cmd]\!]^\sharp(s^\sharp@\ell') \mid \ell = \mathrm{next}(\ell')\}\\ &\cup \{\mathbf{return}^\sharp_{\ell'}(s^\sharp@\ell'') \mid \ell' : x = f(x_1, \ldots, x_n) \wedge \ell = \mathrm{next}(\ell') \wedge \ell'' \in \mathrm{returns}(f)\})\end{aligned}$$

Note that, for simplicity, we have abstracted away boolean expressions. We must also add the identity transfer equation between the label ℓ of a sequential composition $\ell : \{\ell' : cmd_1\ ;\ \ldots\}$ and the label ℓ' of its first command. This completes the definition of the semantic operator $\mathbb{F}^\sharp$. The soundness of our definition is ensured by the following theorem:

Theorem 2. $\mathbb{F} \circ \gamma \subseteq \gamma \circ \mathbb{F}^\sharp$.

The system of fixpoint equations obtained in this way can be solved by applying efficient iteration strategies [2]. In particular, it is not necessary to apply the widening operations at each control point, but only at some points which cut the cycles in the dependency graph.

Example 4. We apply the analysis to the program of Example 1 by using Karr's abstract numerical domain [17] and the results of Steensgaard's analysis described in Example 3. Then, at program point `120`, we obtain the following abstract memory configuration:

$$\begin{aligned}&(n_1, \mathtt{next}, n_1) \mapsto \{= \mathbf{t} + 1\}\\ &(n_1, \mathtt{value}, n_2) \mapsto \{= \mathbf{t}\}\\ &(n_3, \Box, n_2) \mapsto \{= 0, \mathbf{i} = \mathbf{t}\}\end{aligned}$$

together with the following abstract environment:

$$\{\mathtt{l} = 1, \mathtt{t} = 0\}$$

This means that the analysis has been able to discover the *exact* aliasing relationships holding at program point `120`. As far as we know this is the only alias analysis which is able to give this kind of results in the presence of lists and arrays. □

6 Conclusion

In this paper we have constructed a flow-sensitive alias analysis by Abstract Interpretation which can infer position-dependent aliasing relationships. Our work elaborates on the existing literature in this domain by providing a simpler model of the memory which can cope with recursive data structures and arrays within the same framework. This model also tends to be less costly than the ones based on an equivalence relation [11,24]. An interesting aspect from the theoretical viewpoint is that we have designed a technique to solve a entirely symbolic problem in an purely arithmetic framework. Even the previous nonuniform alias analyses which used abstract numerical domains were heavily relying on symbolic algorithms.

We will now explore the scaleability of this model to large-size programs. We are rather confident in the results since flow-sensitive analyses based on numerical domains have shown their ability to handle large programs, which is confirmed by the recent apparition of commercial tools [20]. We will also investigate more sophisticated abstractions of timestamps. In particular, we should encode the call stack more precisely in order to discover nonuniform aliasing relationships which are created recursively. We would then obtain an analysis framework which could be able to analyse recursive data structures and arrays without any restriction on the computation patterns.

Acknowledgement. The author would like to thank the anonymous referees for useful comments on a first version of this paper.

References

1. L.O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994.
2. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the International Conference on Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, pages 128–141, 1993.
3. M. Burke, P. R. Carini, J.-D. Choi, and M. Hind. Interprocedural pointer alias analysis. Technical report, IBM Research Report, 1997.
4. P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, 1981.
5. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth Conference on Principles of Programming Languages POPL'79*, pages 269–282. ACM Press, 1979.
7. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, August 1992.
8. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth Conference on Principles of Programming Languages*. ACM Press, 1978.
9. A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *POPL 1990*, pages 157–168, 1990.
10. A. Deutsch. A storeless model of aliasing and its abstraction using finite representations of right-regular equivalence relations. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 2–13. IEEE Computer Society Press, 1992.
11. A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*. ACM Press, 1994.
12. R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Porceedings of POPL 1996*, pages 1–15, 1996.
13. P. Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–190, 1989.
14. P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT'91*, volume 493. Lecture Notes in Computer Science, 1991.
15. M. Hind and A. Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39(1):31–55, 2001.
16. H.B.M Jonkers. Abstract storage structures. In De Bakker and Van Vliet, editors, *Algorithmic languages*, pages 321–343. IFIP, 1981.
17. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, pages 133–151, 1976.
18. F. Masdupuy. Using abstract interpretation to detect array data dependencies. In *Proceedings of the International Symposium on Supercomputing*, 1991.
19. A. Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO II*, volume 2053 of *LNCS*, pages 155–172. Springer-Verlag, May 2001.
20. PolySpace Technologies. `http://www.polyspace.com`.
21. Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *TOPLAS*, 20(1):1–50, 1998.
22. B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Proceedings of the 1996 International Conference on Compiler Construction*, volume 1060 of *LNCS*, pages 136–150, 1996.
23. A. Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *Proceedings of the Third International Static Analysis Symposium SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 366–382. Springer-Verlag, 1996.
24. A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Science of Computer Programming*, 35(2):223–248, 1999.

Representing and Approximating Transfer Functions in Abstract Interpretation of Hetereogeneous Datatypes

B. Jeannet

INRIA–IRISA

Abstract. We present a general method to combine different datatypes in Abstract Interpretation, within the framework of verification of reactive system. We focus more precisely on the efficient representation and approximation of the transfer functions involved in the abstract fixpoint computations. The solution we propose allows to tune smoothly the necessary tradeoff between accuracy and efficiency in the analysis.

1 Introduction

We are interested in the verification by Abstract Interpretation [8] of reactive systems with infinite state space, generated by a set of variables of different types (enumerated types, integers, FIFO queues, ...). Combining different datatypes in Abstract Interpretation is however difficult, especially in the case where a *relational* abstract domain is needed and when some of the involved datatypes have an infinite domain. For instance, how to represent efficiently the logical relation $b \Rightarrow (x < 0) \vee (x > 10)$, where b is a Boolean and x a real ? And, during a fixpoint computation, how should we compute its image by the vector of transition functions $\begin{pmatrix} b \\ x \end{pmatrix} \longmapsto \begin{pmatrix} b \vee (x > 1) \\ \textbf{if } b \wedge (x > 5) \textbf{ then } 2x \textbf{ else } x - 1 \end{pmatrix}$? Efficiency is here a major issue: for instance synchronous programs [12] that have dozens of Boolean and numerical variables are still small programs. Their transition functions can nevertheless involve many conditions.

[17] proposed a solution based on the partitioning of a basic non-relational abstract domain to represent relationship between datatypes. However it did not focus on the efficient representation and computation of the symbolic transition functions of the analysed system, which is the object of this paper. For this purpose [16] describes a technique inspired by compilation techniques of the Lustre synchronous language [13], and applies it to the verification of Lustre programs with both Boolean and numerical variables.

In this paper we generalize this method, formalize it in an abstract interpretation framework and analyse complexity and accuracy issues. This generalization allows to combine any datatypes, provided that suitable abstract domains are available for each of them. This method has lead to considerable performance improvement without being penalized by the loss of precision. For the application we have in mind, which is the analysis of reactive systems, an important

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 52–68, 2002.
© Springer-Verlag Berlin Heidelberg 2002

point is that the *performance problem* is often as important as the *precision* problem.

Related Work. Several generic operators for composing abstract domains have been proposed in the litterature. The *direct* and *reduced product* [9] lead to domains which are non-relational, and where the image $f(x)$ of a concrete element x by the function f is abstracted by the pair $(f^{\alpha_1}(x^{\alpha_1}), f^{\alpha_2}(x^{\alpha_2}))$: the components do not interact during the computation, which leads in our case to a very coarse approximation. In order to solve this latter limitation, the *open product* [7] uses a notion of *query* which allows f^{α_1} for instance to check if a property on x^{α_2} is satisfied. However, it does not allow to split x^{α_2} w.r.t. to the satisfaction of the property. It should also be noted that these products were mainly intended for combining two abstract domains of the same concrete domain, rather than unrelated domains like $\mathbb{B}^n$ and $\mathsf{Pol}(\mathbb{R}^p)$. Only the *tensor product* [21,22] leads to a truly relational domain. It exhibits the nice property that $\wp(D_1)\otimes\wp(D_2) = \wp(D_1\times D_2)$. If we use it to combine the Boolean lattice $\mathbb{B}^n$ with the lattice of convex polyhedra $\mathsf{Pol}(\mathbb{R}^p)$, we get $\mathbb{B}^n\otimes\mathsf{Pol}(\mathbb{R}^p) = \mathbb{B}^n \to \mathsf{Pol}(\mathbb{R}^p)$, which is actually the lattice used in [14]. We however argued in [17] that its complexity in $\mathcal{O}(2^n)$ limits too strongly the number n of Boolean variables taken into account in the analysis. Moreover it is not clear that the tensor product of infinite lattices is computable (consider $\mathsf{Pol}(\mathbb{R}^p) \otimes \mathsf{Reg}(\Sigma)$, where $\mathsf{Reg}(\Sigma)$ is the lattice of regular languages over Σ).

In the verification community, combining enumerated types with an infinite type (counters, clocks, FIFO queues (e.g. [1,15,4]) is common. In many cases however, finite-state variables are entirely put in the control structure and there is only one infinite type: the verification algorithms manipulate then sets of values of the same type. Among the works aimed at combining different types, [18, 19] use BDDs where some variables are associated to numerical constraints to combine Boolean and numerical datatypes. [5,25] uses instead formula in a kind of disjunctive form. These proposals can be viewed as instances of the *disjunctive completion* technique [10] applied to the direct product of the domains associated to each datatype. The common philosophy of these works is to perform *exact* computations as much as possible and to accept approximations mainly in acceleration and/or widening operators. We give a more precise comparison of these works with ours in the section 5.

In the constraint solving community, there exists classical methods to combine decision procedures, for instance [20,23]. However they are usually restricted to quantifier-free formulas, which prevents the computation of postconditions and allows the computation of preconditions only in the case of closed systems (by using substitution). Their use in verification would also require a suitable widening operator.

Outline of the paper. We first present in section 2 the model of the programs we want to verify, and in section 3 our analysis framework. Section 4 discusses briefly how to manipulate transition functions. Section 5 presents the basic methods to compute transfer functions. Section 6 presents our method based on a combina-

tion of the basic methods. Section 7 explains how to perform partial evaluation and its usefulness on a partitioned system. We present in section 8 some experiments before concluding.

2 The Model of the Considered Programs

We consider in this paper programs that have the following structure: they have only a finite set of control points (defined by a "program counter" variable) and a finite set of variables of different types (Boolean and enumerated types, numbers, queues over some alphabet, ...).

Definition 1 (Types, domains & predicates). *We assume a set* $\mathcal{T} = \{t_1, \ldots, t_{|\mathcal{T}|}\}$ *of* variable types*; to each type* $t \in \mathcal{T}$ *is associated:*

- *a domain* D_t*, in which variables of type* t *take their values;*
- *for any* $k \geq 0$*, a set* $\mathcal{C}_t^{(k)}$ *of* elementary predicates $X \subseteq D_t^k$*;*
- *for any* $k \geq 0$*, a set* $\mathcal{E}_t^{(k)}$ *of* elementary functions $D_t^k \to D_t$*;*

The intuition behind this definition is that elementary predicates and functions are easy to abstract and will be used as building blocks to define more complex predicates and functions.

Example 1. In the case of synchronous programs considered in [17,16], we had $\mathcal{T} = \{\text{bool}, \text{int}\}$ and

$$\left\{ \begin{array}{lll} D_{\text{bool}} = \mathbb{B}, & \mathcal{C}_{\text{bool}}^{(k)} = \{b_i, \bar{b_i}\}_{1 \leq i \leq n}, & \mathcal{E}_{\text{bool}}^{(k)} = \mathbb{B}^k \to \mathbb{B} \\ D_{\text{int}} = \mathbb{Q}, & \mathcal{C}_{\text{int}}^{(k)} = \mathsf{LinCons}(\mathbb{Q}^k), & \mathcal{E}_{\text{int}}^{(k)} = \mathsf{LinExpr}(\mathbb{Q}^k) \end{array} \right\}$$

where $\mathsf{LinExpr}(\mathbb{Q}^k)$ is the set of affine expressions and $\mathsf{LinCons}(\mathbb{Q}^K)$ the set of linear constraints on $\mathbb{Q}^K$. In other words, all Boolean predicates are allowed, whereas elementary numerical predicates are linear constraints and elementary numerical functions are affine functions. □

For any vector $\boldsymbol{n} = (n_1, \ldots, n_{|\mathcal{T}|})$ of natural numbers and for any $t_k \in \mathcal{T}$, n_{t_k} denotes n_k. We note $D^{\boldsymbol{n}}$ the set $\prod_{t \in \mathcal{T}} D_t^{n_t}$, which will be the state space of the program. For any element $d \in D^{\boldsymbol{n}}$, d_t denotes its projection on the domain $D_t^{n_t}$. We will consider in the sequel the following kind of *conditional functions*, built by combining elementary predicates and functions.

Definition 2 (Conditional functions). *The set* $\mathcal{F}_t^{(\boldsymbol{n})}$ *of* conditional functions *of type* $D^{\boldsymbol{n}} \to D_t$ *is defined by the syntax:*

$$\begin{array}{l} \text{fun}_t ::= \mathit{expr}_t \mid \textbf{if } \mathit{cond} \textbf{ then } \text{fun}_t \textbf{ else } \text{fun}_t \\ \mathit{expr}_t \in \mathcal{E}_t^{(n_t)} \quad , \quad \mathit{cond} \in \bigcup_{t \in \mathcal{T}} \mathcal{C}_t^{(n_t)} \end{array}$$

Intuitively, conditional functions are elementary functions guarded by predicates on variables of any type. The value of a function of a certain type may depend on

values of other types only through tests on elementary predicates. The semantics of $f \in \mathcal{F}_t^{(\boldsymbol{n})}$ is the obvious one:

$$[\![\textbf{if } c \textbf{ then } f^+ \textbf{ else } f^-]\!] = \lambda d \in D^{\boldsymbol{n}} \ . \ Ite([\![c]\!](d), [\![f^+]\!](d), [\![f^-]\!](d))$$
$$[\![expr_t \in \mathcal{E}_t^{(n_t)}]\!] = \lambda d \ . \ expr(d_t) \quad , \quad [\![cond \in \mathcal{C}_t^{(n_t)}]\!] = \lambda d \ . \ (d_t \in cond)$$

Example 2. In the previous example, if b_1 and b_2 are Boolean variables and x_1, x_2 are integer variables, then $f(\boldsymbol{b}, \boldsymbol{x}) = (\textbf{if } b_1 \wedge b_2 \textbf{ then } x_1 + x_2 \textbf{ else } x_1)$ is a valid conditional function, but $g(\boldsymbol{b}, \boldsymbol{x}) = (\textbf{if } b_1 \wedge (x_2 \geq 0) \textbf{ then } x_1 + x_2 \textbf{ else } x_1)$ is not valid; it can however be rewritten $g(\boldsymbol{b}, \boldsymbol{x}) = (\textbf{if } b_1 \textbf{ then } (\textbf{if } (x_2 \geq 0) \textbf{ then } x_1 + x_2 \textbf{ else } x_1) \textbf{ else } x_1)$. The figure 1 shows other examples of conditional functions. □

We define now the structure of the programs we analyse.

Definition 3 (symbolic dynamic system). *A symbolic discrete dynamic system is a nuplet* $(V^s, V^i, Init, Ast, (f^v)_{v \in V}, Final)$ *where:*

- $V^s = \bigcup_{t \in \mathcal{T}} V_t^s$ *is the set of* state variable*, where* V_t^s *denotes the set of state variables of type* t*; we note* $n_t^s = |V_t^s|$ *the number of state variables of type* t*, and* $\boldsymbol{n}^s$ *the corresponding vector;* V^i *is the set of input variables, for which we use similar notations;*
- $Init \in \mathcal{F}_{\text{bool}}^{(\boldsymbol{n}^s)}$ *is a Boolean function describing the set of* initial states*;*
- $Ast \in \mathcal{F}_{\text{bool}}^{(\boldsymbol{n}^s + \boldsymbol{n}^i)}$ *is a Boolean function giving in each state the set of possible inputs; it is called the* assertion *of the program;*
- *For every* $t \in \mathcal{T}$ *and variable* $v \in V_t^s$*,* $f^v \in \mathcal{F}_t^{(\boldsymbol{n}^s + \boldsymbol{n}^i)}$ *is a* transition function *giving the value of the variable* v *in the next state in function of the current value of state and input variables;*
- *for verification purpose,* $Final \in \mathcal{F}_{\text{bool}}^{(\boldsymbol{n}^s)}$ *is a* Boolean function *specifying the set of "bad" states that represents the violation of an* invariance *property.*

Such a symbolic discrete dynamic system describes a ordinay discrete dynamic system $(S, I, Init, Ast, \tau, Final)$ where:

- $S = D^{\boldsymbol{n}^s}$ (resp. $I = D^{\boldsymbol{n}^i}$) is the state (resp. input) space;
- $Init = \{\boldsymbol{s} \in S \mid Init(\boldsymbol{s})\}$ is the set of initial states;
- $Ast = \{(\boldsymbol{s}, \boldsymbol{i}) \in S \times D^{\boldsymbol{n}^i} \mid Ast(\boldsymbol{s}, \boldsymbol{i})\}$ is the assertion;
- $\tau \subseteq S \times S$ is defined by $\tau(\boldsymbol{s}, \boldsymbol{s'}) \iff \exists \boldsymbol{i} : \tau(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s'})$ where $\tau(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s'}) \iff Ast(\boldsymbol{s}, \boldsymbol{i}) \wedge (\boldsymbol{s'} = f(\boldsymbol{s}, \boldsymbol{i})) \iff Ast(\boldsymbol{s}, \boldsymbol{i}) \wedge \bigwedge_{v \in V} s'_v = f^v(\boldsymbol{s}, \boldsymbol{i})$;
- $Final = \{\boldsymbol{s} \in S \mid Final(\boldsymbol{s})\}$ is the set of final states.

Input variables and the assertion formula allow to model the environment of an open system. They also allows to introduce non-determinism in an otherwise deterministic systems. Notice that the formalism presented above corresponds to the low-level semantics of dataflow synchronous languages. Most interpreted automaton formalisms can enter in this framework (for instance the models used in [4,1,6]). Imperative programs could also be described in this way, by using an explicit program counter variable and an explicit call stack variable, although it is probably not the most adequate representation.

3 Analysis Framework and Abstract Interpretation

The analysis we want to perform are computations of reachable and/or coreachable states (i.e., states leading to some *final* condition). The applications we have in mind are the computation of invariants and the verification of safety properties. We are also interested in *semantic slicing*, that is, in computing the set of states that belongs to the executions starting from an initial state and leading to a final state. We remind in this section the framework of [16].

Analysis purpose. We consider a symbolic dynamic system $(V^s, V^i, \mathit{Init}, \mathit{Ast}, (f^v)_{v\in V}, \mathit{Final})$ that generates a discrete dynamic system $(S, I, \mathit{Init}, \mathit{Ast}, \tau, \mathit{Final})$. For $X, Y \subseteq S$, let us note $\mathrm{post} = \lambda X.\{\boldsymbol{s'} \mid \exists \boldsymbol{s} \in X : \tau(\boldsymbol{s}, \boldsymbol{s'})\}$ and $\mathrm{pre} = \lambda Y.\{\boldsymbol{s} \mid \exists \boldsymbol{s'} \in Y : \tau(\boldsymbol{s}, \boldsymbol{s'}\}$ the *postcondition* and *precondition* functions. The set of *reachable* (resp. *coreachable*) states of a system is the set $\mathit{reach} = \mathsf{lfp}(\lambda X.\mathit{Init} \cup \mathrm{post}(X))$ (resp. $\mathit{coreach} = \mathsf{lfp}(\lambda X.\mathit{Final} \cup \mathrm{pre}(X))$), where lfp is the least fixpoint operator. If we want to verify a safety property, *Init* will be the set of initial states of the system, *Final* the set of states violating the property, and we want to check that $\mathit{reach} \cap \mathit{Final} = \emptyset$ or $\mathit{Init} \cap \mathit{coreach} = \emptyset$. If we want to perform slicing, *Init* is the set of starting states of the executions of interest, *Final* the set of goal states, and we want to compute $\mathit{reach} \cap \mathit{coreach}$.

The abstract domain. As the above-mentioned fixpoints, defined on the powerset of states $\wp(S)$, are generally not computable, we use the abstract interpretation theory to approximate them. We suppose that for each type $t \in \mathcal{T}$ and integer $n > 0$ we have a Galois connection $(\wp(D_t^n), \alpha_t, \gamma_t, A_t^{(n)})$. Our requirements for a domain $A(\sqsubseteq, \sqcup, \sqcap, \bot, \top)$ abstracting $\wp(S)$ where $S = D^{\boldsymbol{n}} = \prod_{t\in\mathcal{T}} D_t^{n_t}$ are the following: (i) we want to be compositional: A should be defined from the $A_t^{(n_t)}$'s; (ii) we wish a suitable tradeoff between accuracy and complexity of the abstract operations. We explained in the introduction why the "standard" product operators proposed in the Abstract Interpretation community do not satisfy the second requirement. [17] suggests as a solution to take the direct product $A^{(\boldsymbol{n}^s)} = \prod_{t\in\mathcal{T}} A_t^{(n_t^s)}$ to abstract subsets of S, and to use partitioning on this basic *non-relational* domain to establish relations between different datatypes.

Computation of transfer functions. We need suitable upper-approximations of post and pre operators to solve the fixpoint equations in the abstract domain. The best upper-approximation of these operators is often not computable or is too expensive to compute. In addition an accurate result may be worthless because most of the information it contains will be lost when merging the results with the $\sqcup$ operator on the (partitioned) domain.

The next sections present the main contribution of this paper, which is a framework to represent and compute correct approximations of the transfer functions post and pre.

4 Manipulation of Conditional Functions

We explain briefly here how we represent and manipulate conditional functions $f \in \mathcal{F}_t^{(\boldsymbol{n})}$. These functions are proper binary decision diagrams, where decisions

are taken depending on the truth value of elementary predicates, and where their results are elements of type D_t. So the choice of MTBDDs (Multi-Terminal Binary Decision Diagrams) [2] which is an efficient representation of decision diagrams, is very natural. The use of BDDs and/or MTBDDs, the nodes of which are labelled by predicates (or constraints), has been studied in several papers [18,19,3]. Our own use differs however in one point: we use MTBDDs only to represent the transition functions, whereas the cited references use them also to represent and manipulate sets of states.

We will make use of the following operations on MTBDDs. Let $f(c_1, \ldots, c_n)$ be a MTBDD defined on the atomic predicates $c_1, \ldots, c_n$ and returning values in some set T. The cofactor operation f_l where l is a literal (i.e., an atom or its negation) is the partial evaluation of f w.r.t. the literal l. This operation is trivially extended to conjunctions of literals (or monomials) m and is noted f_m. The *support* $\mathsf{supp}(f)$ is the set of predicates involved in f. $\mathsf{T}(f) \subseteq T$ denotes the set of *terminals* of f. The *guard* of a terminal t in f is the BDD $\mathsf{guard}(t, f)$ defined by $\mathsf{guard}(t, f)(c_1, \ldots, c_n) \Leftrightarrow (f(c_1, \ldots, c_n) = t)$. $\mathsf{path}(f)$ denotes the set of paths (m, t) of f, where f is the conjunction of literals encountered from the root node of f to the terminal t. In the case of a BDD, $\mathsf{path}(f)$ will denote the set of paths leading to *true*. For a BDD f (a MTBDD with $T = \{true, false\}$), its factorization $\mathsf{fact\_bdd}(f)$ is the conjunction $f = m \wedge f_m$ where m the smallest monomial such that $f \Rightarrow m$.

5 An Overview of Solutions for Abstract Transfer Functions

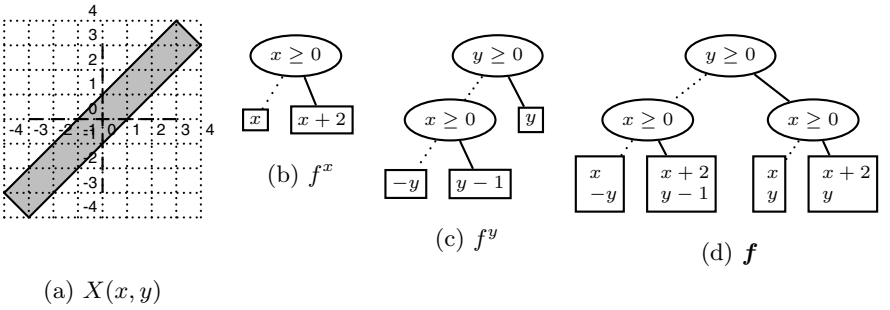
We present here a set of simple methods to approximate transfer functions. We consider a symbolic dynamic system $(V^s, V^i, Init, Ast, (f^v)_{v \in V}, Final)$ generating a discrete dynamic system $(S, I, Init, Ast, \tau, Final)$, where the relation $\tau \subseteq S \times I \times S$ is defined by

$$\tau(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s'}) \iff Ast(\boldsymbol{s}, \boldsymbol{i}) \wedge (\boldsymbol{s'} = f(\boldsymbol{s}, \boldsymbol{i})) \iff Ast(\boldsymbol{s}, \boldsymbol{i}) \wedge \bigwedge_{v \in V^s} s'_v = f^v(\boldsymbol{s}, \boldsymbol{i})$$

For any concrete predicate $x \in \wp(D^{\boldsymbol{n}})$, $[\![x]\!]^\alpha \sqsupseteq \alpha(x)$ will denote in the sequel of the paper some correct approximation of x in $A^{(\boldsymbol{n})}$. As we will later use postcondition and precondition on a partitioned abstract domain, we define the abstract transfer functions in a somehow non-standard way. τ is first extended to the powerset $\tau^\gamma : \wp(S \times I \times S) \to \wp(S \times I \times S)$ with $\tau^\gamma(Z) = \{(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s'}) \in Z \mid \tau(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s'})\}$. Our aim is now to approximate τ^γ with $\tau^\alpha \in A^{(\boldsymbol{n}^s + \boldsymbol{n}^i + \boldsymbol{n}^s)} \to A^{(\boldsymbol{n}^s + \boldsymbol{n}^i + \boldsymbol{n}^s)}$, in order to obtain the following abstract transfer functions:

$$\begin{aligned} \mathrm{post}^\alpha(X, Y) &\triangleq [\![\mathrm{post}(\gamma(X)) \cap \gamma(Y)]\!]^\alpha = \exists \boldsymbol{s} \exists \boldsymbol{i}\ \tau^\alpha(X(\boldsymbol{s}) \sqcap Y(\boldsymbol{s'})) \\ \mathrm{pre}^\alpha(X, Y) &\triangleq [\![\gamma(X) \cap \mathrm{pre}(\gamma(Y))]\!]^\alpha = \exists \boldsymbol{i} \exists \boldsymbol{s'}\ \tau^\alpha(X(\boldsymbol{s}) \sqcap Y(\boldsymbol{s'})) \end{aligned}$$

where $X(\boldsymbol{s})$ and $Y(\boldsymbol{s'})$ are two abstract values.



(a) $X(x, y)$ (b) f^x (c) f^y (d) $\boldsymbol{f}$

Fig. 1. MTBDDs of the transition functions and their product

Best correct upper-approximation. A first solution is to consider the product MTBDD $f = \prod_{v \in V} f^v$, and to consider all of its paths, after having intersected Z with Ast:

$$\tau^{\alpha}_{\text{best}}(Z) = \bigsqcup_{a \in \mathsf{path}(Ast)} \left(\bigsqcup_{(m,\boldsymbol{e}) \in \mathsf{path}(\boldsymbol{f}_a)} [\![a \wedge m]\!]^{\alpha}(\boldsymbol{s}, \boldsymbol{i}) \sqcap [\![\boldsymbol{s'} = \boldsymbol{e}(\boldsymbol{s}, \boldsymbol{i})]\!]^{\alpha} \sqcap Z(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s'}) \right)$$

If we assume that for each variable $v \in V^s$, f^v depends on at most $\#c$ predicates, and we note $\#v = |V^s|$, $\boldsymbol{f}$ have at most $2^{\#c \cdot \#v}$ paths. If we suppose that Ast depends on predicates involved in the f^v, this method requires in the worst case $\mathcal{O}(2^{\#c \cdot \#v})$ atomic operations. We obtain the best correct upper approximation if conjunctions of conditions and elementary functions have best correct approximations, i.e., if $[\![m]\!]^{\alpha} = \alpha([\![m]\!])$ and $[\![\boldsymbol{s'} = \boldsymbol{e}(\boldsymbol{s}, \boldsymbol{i})]\!]^{\alpha} = \alpha([\![\boldsymbol{s'} = \boldsymbol{e}(\boldsymbol{s}, \boldsymbol{i})]\!])$.

Example 3. Let us take $V = V^s = \{x, y\}$, $S = \mathbb{R}^2$, $A = \mathsf{Pol}(\mathbb{R}^2)$, $X = \{x - 1 < y \leq x + 1, -7 \leq x + y \leq 7\}$, $Y = \top$, $Ast = true$ and the transition function defined by:

$$f^x = \textbf{if } x \geq 0 \textbf{ then } x + 2 \textbf{ else } x$$
$$f^y = \textbf{if } y \geq 0 \textbf{ then } y \textbf{ else } (\textbf{if } x \geq 0 \textbf{ then } y - 1 \textbf{ else } -y)$$

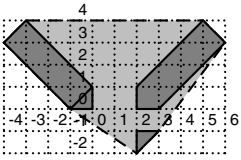
The abstract value X and the MTBDDs of f^x, f^y and their product $\boldsymbol{f}$ are depicted on Fig. 1. We compute

$$\begin{aligned}\text{post}^{\alpha}(X,Y)_{\text{best}} &= \exists x \exists y (\ X \sqcap \{y<0, x<0\} \sqcap \{x'=x, y'=-y\} \ \sqcup\ X \sqcap \{y<0, x \geq 0\} \sqcap \{x'=x+2, y'=y-1\} \ \sqcup \\ &\qquad X \sqcap \{y \geq 0, x<0\} \sqcap \{x'=x, y'=y\} \ \sqcup\ X \sqcap \{y \geq 0, x \geq 0\} \sqcap \{x'=x+2, y'=y\} \\ &= \{y' \leq 4, -1 \leq x'+y' \leq 9, -4 < y'-x' \leq 7, 2x'+3y' \geq -2, 3y'-4x' > -15\}\end{aligned}$$

The result is depicted on Fig. 2.(a). □

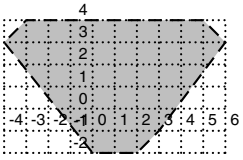
For the sake of simplicity, we assume in the remainder of this section that $Ast = true$.

Cartesian product. To avoid the blow-up of paths due to the product of functions $\prod_{v \in V} f^v$, an alternative is to compute separately the effect of each function f^v on the dimension v' of the state space:

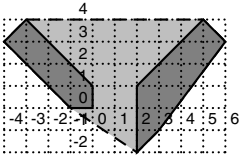


(a) $\mathrm{post}^{\alpha}_{\mathrm{best}}(X, Y)$

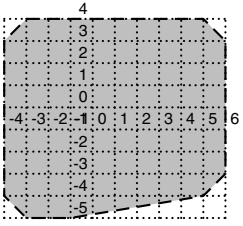
The polygons in dark grey are the images of the polygons obtained by intersecting X with the paths of the product MTBDD $\boldsymbol{f}$. Here their set-union corresponds to the concrete image $\mathrm{post}(X)$.



(b) $\mathrm{post}^{\alpha}_{\mathrm{cart}}(X, Y)$



(c) $\mathrm{post}^{\alpha}_{\mathrm{cart}}(X, Y)$ factorized on $x \geq 0$



(d) $\mathrm{post}^{\alpha}_{\mathrm{cart,abs}}(X, Y)$

Fig. 2. Image of X by different postcondition operators, with $R = \top$

$$\tau^{\alpha}_{\mathrm{cart}}(Z) = \bigsqcap_{v \in V} \left(\bigsqcup_{(m, e^v) \in \mathsf{path}(f^v)} [\![m]\!]^{\alpha}(\boldsymbol{s}, \boldsymbol{i}) \sqcap [\![s'_v = e^v(\boldsymbol{s}, \boldsymbol{i})]\!]^{\alpha} \sqcap Z(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s'}) \right)$$

Using the same notations, the complexity is here $\mathcal{O}(\#v \cdot 2^{\#c})$. Despite the decomposition into a cartesian product, the result is relational because primed variable may be related through their relationship with unprimed variables. However, it is less precise, because lattices are not distributive in general: we only have $X \sqcap (Y_1 \sqcup Y_2) \sqsupseteq (X \sqcap Y_1) \sqcup (X \sqcap Y_2)$.

Example 4. Taking the example 3, we compute

$$\begin{aligned}
&\mathrm{post}^{\alpha}_{\mathrm{cart}}(X,Y) \\
&= \exists x \exists y \Big((X \sqcap \{x<0\} \sqcap \{x'=x\} \sqcup X \sqcap \{x \geq 0\} \sqcap \{x'=x+2\}) \sqcap \\
&\qquad (X \sqcap \{y<0, x<0\} \sqcap \{y'=-y\} \sqcup X \sqcap \{y<0, x \geq 0\} \sqcap \{y'=y-1\} \sqcup X \sqcap \{y \geq 0\} \sqcap \{y'=y\}) \Big) \\
&= \{y' \leq 4, -1 \leq x'+y' \leq 9, -4 < y'-x' \leq 7, 2x'+3y' \geq -2, 3y'-4x' > -15\}
\end{aligned}$$

The result is depicted on Fig. 2.(b). □

Abstracting predicates. Cartesian product allows to tackle the complexity of the postcondition operator which comes from the product of the individual functions f^v. However, what to do when the single functions f^v have themselves complex definitions ? Such a situation happens in synchronous programs. A solution is to simply omit the intersection with predicates involved in MTBDDs and BDDs, and to consider only the sets of their terminals. Together with cartesian

decomposition, we obtain:

$$\tau^{\alpha}_{\text{abs}}(Z) = \bigsqcap_{v \in V} \left(\bigsqcup_{e^v \in \mathsf{T}(f^v)} [\![s'_v = e^v(\boldsymbol{s}, \boldsymbol{i})]\!]^{\alpha} \sqcap Z(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s'}) \right)$$

For the example 3 we get the result depicted on Fig. 2.(d). Assuming that $\#e \leq 2^{\#c}$ is the maximum number of terminals in the MTBDDs, the complexity is here $\mathcal{O}(\#e \cdot \#v)$. In practice we usually have $\#e \ll 2^{\#c}$ and the complexity is considerably reduced. Of course, ignoring predicates garding elementary functions leads to a less precise approximation of τ^{γ} than $\tau^{\alpha}_{\text{cart}}$.

Theorem 1 (Precision and complexity). *The different τ^{α} operators satisfy the inclusion relationship $\tau^{\alpha}_{\text{best}} \sqsubseteq \tau^{\alpha}_{\text{cart}} \sqsubseteq \tau^{\alpha}_{\text{abs}}$ and their complexity is summarized in the following table.*

best	cart	abs
$2^{\#v \cdot \#c}$	$\#v \cdot 2^{\#c}$	$\#v \cdot \#e$

Combination of the above solutions. The possibilities presented above can be refined and combined in order to lead to solutions that exhibit intermediate accuracy and complexity. For the predicate abstraction, an obvious refinement consists in abstracting only a subset of predicates. For the cartesian product, it is possible to partition V into an union of subsets, instead of partitioning it into singletons. For instance, variables which are known to be strongly related should be kept in the same subset in the cartesian decomposition. Also, when the same condition c appears in many functions f^v, it is better to first intersect X with $[\![c]\!]^{\alpha}$ and $[\![\bar{c}]\!]^{\alpha}$ before applying cartesian decomposition. This allows to establish relations between variables depending whether the condition c is satisfied or not and makes the result more precise. We say that we *factorize* the test on the condition c.

Example 5. Taking again example 3, we first factorize the test on the predicate $x \geq 0$ and then apply the cartesian decomposition:

$$\begin{aligned}
&\text{post}^{\alpha}(X,Y) \\
&= \exists x \exists y \Big(\big((X \sqcap \{x<0\} \sqcap \{x'=x\}) \sqcap (X \sqcap \{x<0, y<0\} \sqcap \{y'=-y\} \sqcup X \sqcap \{x<0, y\geq 0\} \sqcap \{y'=y\}) \big) \sqcup \\
&\qquad \big((X \sqcap \{x\geq 0\} \sqcap \{x'=x+2\}) \sqcap (X \sqcap \{x\geq 0, y<0\} \sqcap \{y'=y-1\} \sqcup X \sqcap \{x\geq 0, y\geq 0\} \sqcap \{y'=y\}) \big) \Big) \\
&= \{y' \leq 4, -1 \leq x'+y' \leq 9, -4 < y'-x' \leq 7, 2x'+3y' \geq -2, 3y'-4x' > -15\}
\end{aligned}$$

which is depicted on Fig. 2.(c).

Comparison with existing works. Let us now detail the method of [5] cited in the introduction. In order to represent sets of state and transition functions, [5] uses formulas of the form: $\bigvee_i \bigwedge_{t \in \mathcal{T}} a^t_i$, where the a^t's are formulas of type t. The computation of a postcondition is done by considering the conjunction of the formulas representing respectively the set of states and the transition relation, by putting it in the above-mentioned form, and by eliminating unprimed variables from the obtained formula. The methods [18,19,3] cited in the introduction, instead of the "disjunctive formulas" of [5], uses BDDs, the nodes of

which are labelled by predicates (or constraints). Postcondition computations are performed in a similar way, but with BDD operations, the complexity of which depends on the number of nodes of BDDs.

The obtained postcondition operators are then similar to our best correct approximation method, with the difference that there is no abstract lattice: disjunctions and conjunctions are exact. In both cases, the semantics of predicates is not taken into account by the logical operation on formulas (resp. BDDs) and they can contain unsatisfiable conjunctions (resp. paths). So for emptiness or inclusion test it is necessary to iterates on conjunctions (resp. paths) of the representation and to remove the unsatisfiable ones. In the case of BDDs the complexity of this operation depends obviously on the number of *paths* of the BDD, instead of the number of *nodes*. That means that the sharing provided by BDDs is not so effective here.

6 A Parameterized Heuristic Method

We suggest here a method that combines the previous solutions and offers a parameterized tradeoff between accuracy and efficiency.

6.1 Decomposed Relations and Their Construction

Our abstract relations will be described by a syntaxical structure called *decomposed relation.*

Definition 4 (Decomposed relation). *An* decomposed relation R^V*, indexed by a set* V *of variables, is defined by the following syntax:*

$$
\begin{array}{lll}
ll\ R^V ::= & \mathbf{Assert}(m, R^V) & m = \bigwedge_i c_i \ \textit{with}\ \forall i : c_i \in \mathcal{C} \\
| & \mathbf{Ite}(c, R^V, R^V) & c \in \mathcal{C} \\
| & \mathbf{Cart}(R^{V_1}, \ldots, R^{V_p}) & \{V_1, \ldots, V_p\}\ \textit{being a partition of}\ V; \\
| & \mathbf{Abs}(v, f) & V = \{v\}, f \in \mathcal{F}
\end{array}
$$

Assert is used to intersect the current value with a conjunction of conditions, **Ite** is the if-then-else operator which allows to *open a test* on a condition, **Cart** is the cartesian decomposition operator, and the terminal construct **Abs** allows either to abstract the predicates guarding elementary operations in a conditional function, either to evaluate the effect of an elementary operation. The semantics of a decomposed relation is defined inductively on the table 1. The figure 3 depicts two decomposed relations, where oval boxes denote **Assert**, triangular boxes **Cart**, and diamond boxes **Ite** operators.

Building a correct decomposed relation. Now that we defined a syntaxical structure to describe abstract relations we need an algorithm that generate a decomposed relation R such that $[\![R]\!]^\alpha$ is a correct approximation of τ^γ. We present here such an algorithm. We first assume that $Ast = true$. The idea of the algorithm is to use the **Cart** and **Ite** constructs to factorize the tests on the

Table 1. Semantics of a decomposed relation

R^V	$[\![\cdot]\!]^\alpha : A^{(2\boldsymbol{n}^s+\boldsymbol{n}^i)} \to A^{(2\boldsymbol{n}^s+\boldsymbol{n}^i)}$
$\mathbf{Assert}(m, R)$	$\lambda X \,.\, [\![R]\!]^\alpha(X \sqcap [\![m]\!]^\alpha)$
$\mathbf{Ite}(c, R_t, R_e)$	$\lambda X \,.\, [\![R_t]\!]^\alpha(X \sqcap [\![c]\!]^\alpha) \sqcup [\![R_e]\!]^\alpha(X \sqcap [\![\bar{c}]\!]^\alpha)$
$\mathbf{Cart}(R_1, \ldots, R_p)$	$\lambda X \,.\, \bigsqcap_{1\leq i\leq p}([\![R_i]\!]^\alpha(X))$
$\mathbf{Abs}(v, f)$	$\lambda X \,.\, \bigsqcap_{e\in\mathsf{T}(f)} X \sqcap [\![s'_v = e(\boldsymbol{s}, \boldsymbol{i})]\!]^\alpha$

Algorithm 1 Decomposing the transition relation

Function *decompose*$(V, \boldsymbol{f}, d)$
Parameter: *An integer* $0 \leq \mathrm{dmax}_{\mathrm{ite}} \leq \infty$ *(max. depth of* **Ite** *constructs)*
Input: *A set of variables* V*, a vector* $\boldsymbol{f}$ *of conditional functions, and an integer* d
Output: *A decomposed relation* R

(1) **begin**
(2) **if** $(\forall v \in V : f^v \in \bigcup_t \mathcal{E}_t^{(n_t)})$ **or** $(d = \mathrm{dmax}_{\mathrm{ite}})$ **then**
(3) **return Cart**$(\{\mathbf{Abs}(v, f^v) \mid v \in V\})$
(4) **else**
(5) **let** $c = \mathit{choose\_cond}(V)$ **in**
(6) **let** $V^i = \mathit{select\_var}(c, V)$ **in**
(7) **let** $R^+ = \mathit{decompose}(V^i, \boldsymbol{f}_c^{V^i}, d+1)$ **in**
(8) **let** $R^- = \mathit{decompose}(V^i, \boldsymbol{f}_{\bar{c}}^{V^i}, d+1)$ **in**
(9) **let** $R^o = \mathit{decompose}(V \setminus V^i, \boldsymbol{f}^{V\setminus V^i}, d)$ **in**
(10) **return Cart**$(\mathbf{Ite}(\mathbf{c}, \mathbf{R}^+, \mathbf{R}^-), \mathbf{R^o})$
(11) **end**

predicates that are involved in several transition functions, and to compute the effect of elementary operations, by using **Cart** and **Abs** operators.

Let us detail the algorithm 1. First of all, in line (1), if all the variables in V are associated to elementary functions, they can be directly computed by decomposing them into singletons and using the **Abs** construct. Also, if the maximum depth $d = \mathrm{dmax}_{\mathrm{ite}}$ of **Ite** constructs has been reached, we use the predicate abstraction, instead of going on opening tests with the **Ite** construct.

Otherwise, we should open a test. Line (5) selects a predicate on which a test will be opened, then line (6) selects the variables that will be involved in the two branches of the test. Once this is done, we apply recursively the algorithm. On one hand, lines (7–8) compute the cofactor of the conditional functions of the variables V^i according to the predicate c and apply recursively the algorithm, in order to generate the two branches of the **Ite** construct. On the other hand, line (9) applies recursively the algorithm to the remaining dimensions. Line (10) puts together the results of the recursive calls.

We have still to define the functions *choose_cond* and *select_var*. Our heuristic to select a condition is to choose the one which appears in the greatest number of conditional functions. The algorithm 2 describe the variable selection algo-

Algorithm 2 Selecting the variables to be put in the **Ite** construct

Function *select_var*$(c, V, \boldsymbol{f})$
Parameter: *An integer* $-1 \leq \mathrm{dmax}_{\mathrm{select}} \leq \infty$
Input: *a condition* c*, a set* V *of variable and the associated vector* $\boldsymbol{f}$ *of conditional functions*
Output: *a subset* W *of* V

(1) **begin**
(2) **if** $\mathrm{dmax}_{\mathrm{select}} = -1$ **then**
(3) **return** $\{v\}$*, such that* $c \in \mathsf{supp}(f^v)$
(4) **else**
(5) **let** $W_c = \{v \in V \mid c \in \mathsf{supp}(f^v)\}$ **in**
(6) **let** $W_o = \{v \in V \mid |\mathsf{supp}(f^v)| < \mathrm{dmax}_{\mathrm{select}}\}$ **in**
(7) **return** $W_c \cup W_o$
(8) **end**

rithm. In the case where $\mathrm{dmax}_{\mathrm{select}} \geq 0$, the line (5) selects all the variables, the associated function of which depends on the condition: it is natural to put these functions in the branches of the **Ite** construct. What to do with the other functions ? If we want for instance to obtain the best correct approximation, we have to select *all the remaining variables*. In order to leave opened a full range of possibilities, we use the parameter $\mathrm{dmax}_{\mathrm{select}}$: if a function depends on less than $\mathrm{dmax}_{\mathrm{select}}$ predicates, line (6) will select the associated variable. The special case $\mathrm{dmax}_{\mathrm{select}} = -1$ allows to forbid any factorization of predicates and to obtain the $\mathrm{post}^{\alpha}_{\mathrm{cart}}$ described in section 5.

Satisfying the assertion. If we remove the assumption that $Ast = true$, we have to satisfy it before computing the relation. This is performed by the algorithm 3, which use **Ite** and **Assert** constructs to recursively decompose the MTBDD of Ast. The parameter $\mathrm{dmax}_{\mathrm{assert}}$ allows to bound the depth of the **Ite** constructs. If the line (7) is executed, then the assertion will be only partially satisfied on the corresponding branch.

Theorem 2 (Correctness). *For any decomposed relation* R *constructed by the algorithm 3,* $[\![R]\!]^{\alpha} \sqsupseteq \tau^{\alpha}_{\mathrm{best}}$

6.2 Complexity and Accuracy Results

Let us denote by $R_{(a,i,s)}$ the decomposed relation generated by the algorithm 3 with the parameters $\mathrm{dmax}_{\mathrm{assert}} = a$, $\mathrm{dmax}_{\mathrm{ite}} = i$, $\mathrm{dmax}_{\mathrm{select}} = s$. First, notice that by adjusting these parameters, it is possible to obtain some of the postcondition operators described in section 5; for instance:

$$\begin{aligned} [\![R_{(\infty,\infty,\infty)}]\!]^{\alpha} &= \tau^{\alpha}_{\mathrm{best}} \\ [\![R_{(\infty,\infty,-1)}]\!]^{\alpha} &= \tau^{\alpha}_{\mathrm{cart}} \\ [\![R_{(0,0,-1)}]\!]^{\alpha} = [\![R_{(0,0,0)}]\!]^{\alpha} &= \tau^{\alpha}_{\mathrm{abs}} \end{aligned}$$

Algorithm 3 Satisfying the assertion

Function *make*$(a, \boldsymbol{f}, d)$
Parameter: *An integer* $\mathrm{dmax}_{\mathrm{assert}}$
Input: *An assertion* a*, a vector* $\boldsymbol{f}$*, and an integer* d
Output: *A decomposed relation* R

```
(1)  begin
(2)    let (m, a') = factorize_bdd(a) in
(3)    match a' with
(4)    Δ(true) → return Assert(m, decompose(V, f_m, 0))
(5)    Δ(c, a⁺, a⁻) →
(6)      if d = dmax_assert then
(7)        return Assert(m, decompose(V, f_m, 0))
(8)      else
(9)        let R⁺ = make(a⁺, f_{m∧c}, d + 1) in
(10)       let R⁻ = make(a⁻, f_{m∧c̄}, d + 1) in
(11)       return Assert(m, Ite(c, R⁺, R⁻))
(12) end
```

Theorem 3. *The following relationships between decomposed relations holds:*

$$\begin{gathered}[\![R_{(a+1,\infty,\infty)}]\!]^\alpha \sqsubseteq [\![R_{(a,\infty,\infty)}]\!]^\alpha \\ [\![R_{(a,i+1,s)}]\!]^\alpha \sqsubseteq [\![R_{(a,i,s)}]\!]^\alpha \\ [\![R_{(a,\infty,\infty)}]\!]^\alpha \sqsubseteq [\![R_{(a,i,1)}]\!]^\alpha \sqsubseteq [\![R_{(a,i,0)}]\!]^\alpha \sqsubseteq [\![R_{(a,i,-1)}]\!]^\alpha\end{gathered}$$

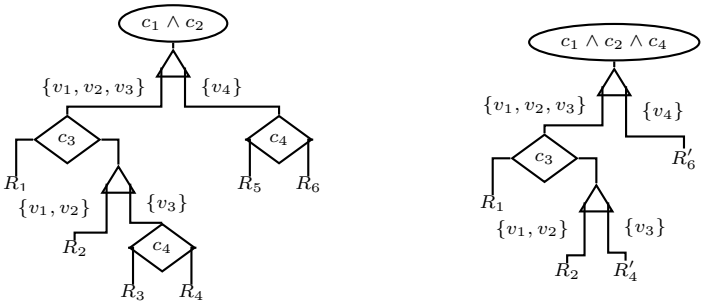
Moreover, the size of $R_{(a,i,s)}$ *is in* $\mathcal{O}(2^{\min(a+d,\#c)} \cdot \#e \cdot \#v)$ *for* $s \leq 1$*, and in* $\mathcal{O}(2^{\min(a+d,\#c+(\#v-1)\cdot\min(s,\#c))} \cdot \#e \cdot \#v)$ *for* $s \geq 2$*.*

The arguments used in the proof are mainly the monotonicity of the operators defining decomposed relations and the property $(X \sqcap Y_1) \sqcup (X \sqcap Y_2) \sqsubseteq X \sqcap (Y_1 \sqcup Y_2)$. Proving more precise results is difficult, and perhaps impossible, because the modification of one parameter can completely change the structure of the recursive calls in the generation algorithm.

7 Specialization of a Decomposition and Partitioned Systems

We show here how to use the assertion to perform a partial evaluation of a decomposed relation on both origin and destination predicates, and how to take advantage of it when a partitioned domain is used.

Principle. Let us consider a decomposed relation R, two abstract values $P(\boldsymbol{s})$ and $Q(\boldsymbol{s'})$, and let us evaluate $[\![R]\!]^\alpha(P \sqcap Q)$. Consider an $\mathbf{Ite}(c, \ldots)$ construct, and let us note m the conjunction of conditions encountered in the **Assert** and **Ite** constructs from the root to this construct. Suppose that during the recursive evaluation following the table 1, the result of the negative branch of the $\mathbf{Ite}(c, \ldots)$ construct returns $\bot$. That means $\boldsymbol{s} \in P$ and $\boldsymbol{s'} \in Q$ are related only if $m \wedge c$



(a) Before specialization

(b) After specialization

Fig. 3. Specialization of a decomposed relation

is satisfied. We can then strengthen the assertion with this information, and recompute a new decomposition. We obtain then a decomposed relation R wich gives a correct approximation of τ for any $X(\boldsymbol{s}) \sqsubseteq P(\boldsymbol{s})$ and $Y(\boldsymbol{s'}) \sqsubseteq Q(\boldsymbol{s'})$.

Example 6. For instance, on the figure 3(a), if we suppose that this happens on the $\mathbf{Ite}(\mathbf{c_4}, \mathbf{R_5}, \mathbf{R_6})$ construct, we get a new assertion $c_1 \wedge c_2 \wedge c_4$ and we can generate the decomposed relation depicted on the figure 3(b). Here the depth of **Ite** constructs is decreased by 1 above R_4 and R_6, which means that less predicates will be abstracted in R'_4 and R'_6.

The general procedure consists in evaluationg $[\![R]\!]^\alpha$ on $P \sqcap Q$, to compute the disjunction g of all the conjunction of conditions leading to $\perp$, to strengthen the formula Ast with $\neg g$, and to recompute a new decomposed relation.

Interest of the specialization. If we fix the parameters (a, i, s), strengthening the assertion by specializing a decomposed relation $R_{(a,i,s)}$ on $P(\boldsymbol{s}) \sqcap Q(\boldsymbol{s'})$ is very interesting, because the algorithm 1 is applied to simpler conditional functions. As a consequence, the number of abstracted predicates is decreased and the factorization can be more effective. So, with a constant complexity, the accuracy is improved.

It also provides a way to compute necessary conditions for a concrete state in P to lead to a concrete state in Q in an incremental way. We can repeat the process of evaluating $[\![R]\!]^\alpha(P \sqcap Q)$, strengthening the assertion and computing a new decomposition. The iteration will converge because only a finite number of predicates are involved in the conditional functions.

Use of specialization on partitioned systems. On a partitioned abstract lattice, where a set of concrete states is abstracted by a n-uplet of disjoint abstract values [16], specialization can be used by specializing assertions and decomposed relations on edges linking two abstract classes. Consider a finite partition $\mathcal{K} = \{P^{(k)} \in A \mid k \in K\}$ of S into abstract values. For each pair of classes (k_1, k_2) we can consider an assertion $Ast^{k_1 \rightsquigarrow k_2}$ and a decomposed relation $R^{k_1 \rightsquigarrow k_2}$ specialized on $P^{(k_1)}(\boldsymbol{s}) \sqcap P^{(k_2)}(\boldsymbol{s'})$. During forward analysis, postconditions will be computed along edges with $\text{post}^{k_1 \rightsquigarrow k_2} = \lambda X \sqsubseteq$

Table 2. Experiment

N	bool var state/input	num var state/input	numerical constraints	Assertion (nodes,paths)
2	7/3	6/0	23	(15,55)
3	10/4	8/0	34	(26,239)
4	13/5	10/0	45	(45,1055)

Conditional functions (for each state variable):
$\approx$ 10 nodes, $\approx$ 20 paths

(a) Program statistics

N	2			3			4		
(a,i,s)	time	ana	ite	time	ana	ite	time	ana	ite
$(1,1,2)$	3.9	1	6	12.0	1	9	107.6	1	11
$(2,2,2)$	6.2	2	25	15.6	2	39	34.4	2	52
$(3,3,2)$	9.0	2	53	28.7	2	118	64.0	2	157
$(5,5,2)$	14.8	2	119	65.9	2	276	208.9	2	588
$(\infty,\infty,2)$	13.9	2	138	75.1	2	478	359.2	2	1424

(b) Property 2, lower bound

N	2		3		4	
(a,i,s)	time	ana	time	ana	time	ana
$(1,1,2)$	failure		failure		failure	
$(2,2,2)$	22.6	4	93.2	4	220.0	4
$(3,3,2)$	31.0	4	136.6	4	479.2	4
$(5,5,2)$	40.2	4	212.4	4	858.3	4
$(\infty,\infty,2)$	43.3	4	291.0	4	swap !	–

(c) Property 2, upper bound

$P^{(k)}.[\![R]\!]^{\alpha}(X(\boldsymbol{s}) \sqcap P^{(k_2)}(\boldsymbol{s'}))$, and similar precondition operators will be used for backward analysis.

Assertions associated to edges can also be used to refine a partition. Indeed, comparing two assertions $Ast^{k \rightsquigarrow k_1}$ and $Ast^{k \rightsquigarrow k_2}$ allows to separate those states in $P^{(k)}$ wich can only lead to k_1 or k_2. Such information is the basis of several partition refinement techniques, in abstract interpretation [16,11] or in the combination of deduction and model-checking [24].

Last but not least, our technique allows to link the precision of the transfer functions to the precision induced by the partition: when a partition is refined, specialization becomes stronger and decomposed relations are more precise.

8 Experimental Evaluation

We implemented our technique in the tool NBAC [16], which perform verification of LUSTRE programs [13] having both Boolean and numerical variables. The algorithms implemented in this tool include forward and backward analysis, automatic partition refinement and interleaved iterations of these.

We experimented it on a railway control system described in [16]. Table 2(a) gives information on the input program, where N is the number of trains. The assertion, which models the environment and from which comes in a great part the complexity of the system, states basically that the trains obey the orders they are given. Tables 2(b) and 2(c) gives the verification results for the property 2 defined in [16]. We give for different values of the parameters the time in seconds, the number of analysis cycles followed by partition refinement cycles, and the average number of **Ite** per decomposed relation (only in table 2(b)). We used the standard Boolean lattice for the Boolean variables (i.e., Boolean variables are not abstracted) and the convex polyhedra lattice $\mathsf{Pol}(\mathbb{Q}^n)$ for the numerical variables. We were able to verify the property only for $s = \mathrm{dmax}_{\mathrm{select}} \geq 2$. The

tables shows that on this example a small bound on the maximum depth of nested **Ite** operators still allows the verification to be successful, and at a very cheap cost compared to the cost observed when this depth is not bounded (row indexed by $(\infty, \infty, 2)$). Observe also that the use of less precise operators does not require a higher number of refinement steps to prove the property, which shows that a greater precision is not needed for the proof.

9 Conclusion

We proposed a generic framework to combine different datatypes in Abstract Interpretation of reactive systems. Our solution is based on the assumption that transition functions of the analysed system can be represented by conditional functions as defined in definition 2. This quite liberal assumption allows us to design a generic method to analyse and verify such programs, which relies on the following ingredients:

- Combining abstract lattices devoted to each datatype into a global abstract lattice with the very simple *direct product* operator;
- Using *partitioning* on the previous lattice to improve its precision;
- Using *decomposed relations* to compute transfer functions and to adapt their precision with the precision induced by the partitioning.

The paper focuses more precisely on the last point. Our preliminary experiments shows the usefulness of approximating transfer functions in a flexible and parametrized way: it can make the global method considerably more efficient. Our future plans are to experiment the combination of other datatypes and/or abstract lattices with this method.

References

1. P. Aziz Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In *Computer Aided Verification, CAV'98*, volume 1427 of *LNCS*, July 1998.
2. R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, April 1997.
3. G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In *Computer Aided Verification, CAV'99*, volume 1633 of *LNCS*, July 1999.
4. B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Computer Aided Verification, CAV'96*, volume 1102 of *LNCS*, July 1996.
5. T. Bultan, R. Gerber, and C. League. Composite model checking: Verification with type-specific symbolic representations. *ACM Transactions on Software Engineering and Methodology*, 9(1), January 2000.
6. T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *Computer Aided Verification, CAV'97*, volume 1254 of *LNCS*, June 1997.

7. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combination of abstract domains for logic programming: open product and generic pattern construct. *Science of Computer Programming*, 38, 2000.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages, POPL'77*, Los Angeles, January 1977.
9. 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.
10. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 1992.
11. R. Giacobazzi and E. Qintarelli. Incompleteness, counterexamples, and refinement in abstract model-checking. In *Static Analysis Symposium, SAS'01*, volume 2126 of *LNCS*, July 2001.
12. N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.
13. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
14. N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2), August 1997.
15. T. Henzinger, P.-H. Ho, and H. Wong-Toï. HyTech: A Model Checker for Hybrid Systems. In *Computer Aided Verification, CAV'97*, number 1254 in LNCS, June 1997.
16. B. Jeannet. Dynamic partitioning in linear relation analysis. Application to the verification of reactive systems. *Formal Methods in System Design.* 40 pages, to appear, available as a BRICS Research Report `http://www.brics.dk/RS/00/38`.
17. B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium, SAS'99*, volume 1694 of *LNCS*, Venezia (Italy), September 1999.
18. C. Mauras. Calcul symbolique et automates interprétés. Technical Report 10, IRCyN, November 1996.
19. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Computer Science Logic*, The IT University of Copenhagen, Denmark, September 1999.
20. G. Nelson and C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2), 1979.
21. F. Nielson. Tensor products generalize the relational data flow analysis method. In *Fourth Hungarian Computer Science Conference*, 1985.
22. F. Nielson and H. R. Nielson. The tensor product in wadler's analysis of list. *Science of Computer Programming*, 22, 1994.
23. R. Shostak. Deciding combination of theories. *Journal of the ACM*, 31(1), 1984.
24. H. Sipma, T. Uribe, and Z. Manna. Model checking and deduction for infinite-state systems. In *Logical Perspectives on Language and Information.* CSLI publications, 2001.
25. T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representations. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'01*, volume 2031 of *LNCS*, April 2001.

Automated Verification of Concurrent Linked Lists with Counters*

Tuba Yavuz-Kahveci and Tevfik Bultan

Department of Computer Science, University of California,
Santa Barbara, CA 93106 USA
{tuba,bultan}@cs.ucsb.edu

Abstract. We present an automated verification technique for verification of concurrent linked lists with integer variables. We show that using our technique one can automatically verify invariants that relate (unbounded) integer variables and heap variables such as $head \neq null \rightarrow numItems > 0$. The presented technique extends our previous work on composite symbolic representations with shape analysis. The main idea is to use different data structures such as BDDs, arithmetic constraints and shape graphs as type specific symbolic representations in automated verification. We show that polyhedra based widening operation can be integrated with summarization operation in shape graphs to conservatively verify properties of concurrent linked lists.

1 Introduction

Manipulation of linked lists is tricky. So is concurrent programming. In this paper we present a technique for automated verification of concurrent linked list specifications. Since we allow unbounded variables (such as integers) verification problem in general is not decidable. Our algorithm computes conservative approximations to the the reachable states of the input system to verify or falsify invariants. When we verify an invariant it is guaranteed to hold, and when we falsify an invariant the invariant is definitely violated. However, we may not be able to verify all the invariants that actually hold, and we may not be able to falsify all the invariants that are actually violated.

We use symbolic forward fixpoint computations in which values of boolean and enumerated variables are encoded using BDDs, values of integer variables are encoded using linear arithmetic constraints, and values of heap variables and configurations of the heap are encoded using shape graphs. Our symbolic manipulator can compute the set of reachable states of an input system (where each state corresponds to valuations of all the variables in the system and the configuration of the heap) given a finite bound on the number of execution steps, i.e., it can compute a lower approximation to the reachable states of the system. This is done by iteratively applying the post-condition operation starting from

* This work is supported in part by NSF grant CCR-9970976 and NSF CAREER award CCR-9984822.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 69–84, 2002.
© Springer-Verlag Berlin Heidelberg 2002

the initial states. This lower approximation can be used to search for errors. More interestingly, our verification algorithm can compute an upper approximation to the set of reachable states and demonstrate the absence of errors by showing that the invariants are satisfied by all the states in the upper approximation. We compute an upper approximation to the set of reachable states using two concepts: 1) summarization nodes in shape graphs, and 2) widening operation on polyhedra. Together, these operations force the least fixpoint computation for the set of reachable states to converge to an upper approximation. We demonstrate that the approximate fixpoint computation is precise enough to prove interesting properties on some example concurrent linked list specifications. Although the technique we present for computing a lower approximation to the reachable states generalizes to heap objects with multiple selector fields, the summarization technique we present (hence, the upper approximation of reachable states) is customized for heap objects with a single selector field.

Our work in this paper combines and extends previous work on: 1) *Shape analysis:* Shape analysis algorithm we use in this paper builds on the previous work in this area [SRW98,DRS00,LAS00,LARSW00,Yah01]. However, we extend the shape graph representation by keeping an integer count for each summary node that keeps track of the number of concrete heap nodes represented by the corresponding summary node. 2) *Verification with arithmetic constraint manipulation:* Arithmetic constraints represented as polyhedra have been used in analyzing real-time and concurrent systems [AHH96,BGP99,HRP94]. The techniques we use for integer constraint manipulation are based on [CH78] and [BGP99]. 3) *Model checking with composite representations:* Our earlier work on composite representations concentrated on using BDDs and polyhedral representations together as a symbolic representation to verify infinite state systems with integer variables efficiently [YKTB01]. Here, we extend this work by adding shape graphs as a new symbolic representation.

Our main contribution in this paper is to show a technique that merges shape analysis and verification based on arithmetic constraints under the composite symbolic representation framework. It is not possible to verify the types of systems we are analyzing using the techniques presented in earlier work. The approach used in [LAS00,LARSW00,Yah01] can be extended to handle some predicates on integers but this will require addition of instrumentation predicates and formulas to show how integer predicates change based on single-step execution. Our approach is completely automatic, the key invariants that relate the number of nodes in a linked list to the integer variables in the system are automatically discovered by our verification algorithm. Our tool discovers these invariants using integer counters which keep track of the number of concrete heap nodes represented by summary nodes and using polyhedral widening operation. These invariants in turn imply the relationships between the values of the heap variables and the values of the integer variables. We integrated the shape analysis algorithm presented in this paper to our tools Composite Symbolic Library [YKTB01] and Action Language Verifier [BYK01] using a BDD encoding

```
module main()
 heap {next} head, tail, add, get, newHead;
 boolean mutex; integer numItems;
 initial: head=null and tail=null and numItems=0 and mutex;
 module put()
  enumerated pc {create, setNull, checknull, updateTail};
  initial: pc=create and add=null;
  put1: pc=create and mutex and !mutex' and add'=new and pc'=setNull;
  put2: pc=setNull and add'.next=null and pc'=checknull;
  put3: pc=checknull and numItems=0 and tail'=add and
        head'=add and numItems'=1 and mutex' and pc'=create;
  put4: pc=checknull and numItems>0 and tail'.next=add and pc'=updateTail;
  put5: pc=updateTail and tail'=add and
        numItems'=numItems+1 and mutex' and pc'=create;
  put: put1 | put2 | put3 | put4 | put5;
 endmodule
 module take()
  enumerated pc {copyHeadNext, setNextnull, updateTail};
  initial: pc=copyHeadNext and get=null;
  take1: pc=copyHeadNext and mutex and numItems>0 and !mutex' and
         newHead'=head.next and get'=head and pc'=setNextnull;
  take2: pc=setNextnull and get'.next=null and head'=newHead and
         numItems'=numItems-1 and pc'=updateTail;
  take3: pc=updateTail and newHead=null and tail'=null
         and mutex' and pc'=copyHeadNext;
  take4: pc=updateTail and newHead!=null and mutex' and pc'=copyHeadNext;
  take:  take1 | take2 | take3 | take4;
 endmodule
 main: put() | put() | take() | take() ;
 spec: invariant((head=null and tail=null) <=> numItems=0)
endmodule
```

Fig. 1. Action Language specification of a concurrent linked list

of the shape graphs. We verified several properties of three concurrent linked list specifications.

In Section 2, we present a concurrent linked list example in our input specification language. We discuss the composite symbolic representation in Section 3. In Section 4, we discuss the integration of shape graphs to the composite symbolic representation and in Section 5, we discuss summarization and widening operations. We report our experimental results in Section 6 and give our conclusions in Section 7.

2 A Concurrent Linked List Example in Action Language

As an example system consider the concurrent linked list specification given in Action Language in Figure 1. Action Language is a specification language

for modular specification of reactive software systems which allows both asynchronous and synchronous compositions (in this paper we are focusing on verification of asynchronous systems) [BYK01]. Action Language can be thought of as an intermediate language for verification. One can automatically translate a concurrent linked list implementation to an Action Language specification and verify it using our verification tools. Alternatively, one can specify a concurrent linked list in Action Language, verify it, and then automatically generate an implementation from the Action Language specification (which is the approach we took in [YKB02]). In this paper we will concentrate on verification procedures for concurrent linked lists specified in Action Language. These techniques can be extended to verification of concurrent linked lists in programming languages such as Java using one of the approaches above.

An Action Language specification consists of a set of hierarchical module definitions and a set of properties to be verified about the specification. Semantically, each module corresponds to a transition system with a set of states, a set of initial states and a transition relation. In Figure 1, module `main` has two submodules `put` and `take`. Submodule `put` models a process that adds a new item to the linked list and submodule `take` models a process that removes an item from the linked list. Each submodule has a local variable `pc` denoting the program counter. Each module instantiation creates a new instantiation of the local variables of that module. Variables of the `main` module correspond to global variables. The variables `head`, `tail`, `add`, `get`, and `newHead` are heap variables with a selector (i.e., field) called `next`. We are assuming that all selectors of heap objects are pointers to heap (i.e., we are ignoring data members such as integers or booleans). The `main` module has an integer variable `numItems` which keeps track of the number of items in the linked list, and a boolean variable `mutex` which is used as a semaphore.

An initial expression defines the set of initial states of a module. The initial expressions are written as state formulas. We call an expression a *state formula* if it contains no primed variables. In Figure 1, initially, `head` and `tail` are equal to `null`, `numItems` is equal to 0, and the `mutex` is `true`. Initial expressions of submodules `take` and `put` define the initial values of their program counters.

Behavior of a module (i.e., its transition relation) is defined using a module expression. A module expression (which starts with the name of the module) is written by combining its actions and submodules using asynchronous (denoted by `|`) and/or synchronous (denoted by `&`) composition operators. For instance, module `put` is defined in terms of asynchronous composition of its actions `put1`, `put2`, and so on. Module `main` (which defines the transition relation of the overall system) is defined in terms of asynchronous composition of two instantiations of its submodule `put` and two instantiations of its submodule `take`. This models the scenario where there are two processes that add new items to the list and two processes that remove items from the list.

Each action in Action Language defines a single execution step. The actions are specified as transition formulas. We call an expression a *transition formula* if it contains one or more primed variables. In a transition formula, primed

variables denote the next-state values for the variables and unprimed variables denote the current-state values. For instance, action `put3` in module `put` states that when `pc=checkNull` and `numItems=0` , then in the next state, `tail` and `head` will be set to `add`, `numItems` will be set to 1 and `mutex` will be set to true and `pc` will be set to `create`.

Asynchronous composition of two actions is defined as the disjunction of their transition relations. However, we also assume that an action preserves the values of the variables which are not modified by itself.

The final line in the definition of the `main` module indicates that we wish the Action Language Verifier to check if the state formula `(head=null and tail=null) <=> numItems=0` is an invariant of this system.

The expressions in Action Language are written as *composite formulas*. A composite formula consists of boolean, integer, and heap formulas connected by logical connectives [YKB02]. Integer formulas consist of linear arithmetic constraints on integer variables and constants connected by logical connectives. Similarly, boolean and heap formulas only refer to boolean and heap variables and constants, respectively.

As mentioned above, a composite formula is a state formula if it contains no primed variables and it is a transition formula if it contains one or more primed variables. The initial condition and the invariants are specified using state formulas and the actions are specified as transition formulas. We will put the following restrictions on the transition formulas:

- A primed heap variable can only appear once in a transition formula in an equality.
- If a heap variable appears primed in a transition formula it can only appear unprimed once in the same transition formula.
- A transition formula is in the form of a conjunction of a heap formula, a boolean formula and an integer formula.
- The heap subformula of a transition formula is a conjunction of a guard which is a heap state formula and an update formula. The update formula is a conjunction of equalities.

3 Verification with Composite Symbolic Representation

A composite symbolic representation maps each variable in the input specification to a *basic symbolic representation*. The sets of system states and transitions are represented as a disjunction of conjunctions of these type-specific, basic symbolic representations. Hence, given a composite formula A its *composite representation* is in the form

$$A \equiv \bigvee_{i=1}^{n} \bigwedge_{t \in T} a_{it}$$

where a_{it} denotes the basic symbolic representation of type t in the ith disjunct, and n and T denote the number of disjuncts and the set of basic symbolic representations, respectively. In the verification algorithms we will present in this

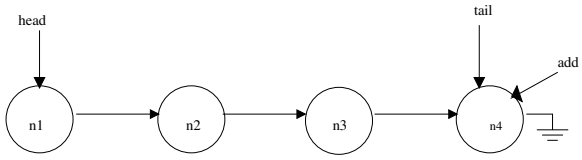


Fig. 2. A linked list with four nodes.

paper $T = \{boolean, integer, heap\}$. We call each disjunct $\wedge_{t \in T} a_{it}$ a *composite atom*.

Each atomic event in the input specification is conjunctively partitioned where each conjunct specifies the effect of the event on the variables represented by a basic symbolic representation. The post-condition computations are computed independently for each symbolic representation by exploiting the conjunctive partitioning of the atomic events.

In this paper we add a new basic symbolic representation for heap variables to the Composite Symbolic Library [YKTB01]. We use sets of shape graphs as a symbolic representation to encode configurations of the heap. The states of boolean and enumerated variables are represented as boolean logic formulas, encoded as BDDs using the Colorado University Decision Diagram Package (CUDD) [CUD]. The states of integer variables are represented as Presburger arithmetic formulas encoded as sets of polyhedra using the Omega Library [Ome]. Basic set operations intersection, union, complement, and subset, equality, and emptiness checks on the disjunctive composite representation can be implemented using corresponding operations on basic symbolic representations as discussed in [YKTB01].

One can use post-condition computation to compute the set of reachable states of a system. The (exact) reachable state-space of a system is the least fixpoint $RS \equiv \mu x \ . \ I \ \vee \ \textsc{post}(x, R)$, where R is the transition relation of the system. We can compute this fixpoint by starting the iteration from the set of initial states and iteratively applying the post-condition computation. However, in an infinite state system convergence is not guaranteed. Even though the exact fixpoint computations are not guaranteed to converge, they may converge to the least fixpoint for some systems. If we cannot compute the set of reachable states RS exactly we can try to compute approximations to it RS^+ or RS^-, such that, $RS^- \subseteq RS \subseteq RS^+$. To verify an invariant we can compute RS^+ and check if all the states in RS^+ satisfy the invariant. To falsify an invariant we can compute RS^- and check if there is a state in RS^- which violates the invariant.

4 Integrating Shape Analysis to Composite Symbolic Representation

In this section we will explain the integration of the heap representation to the composite symbolic representation. We represent the set of states of the heap

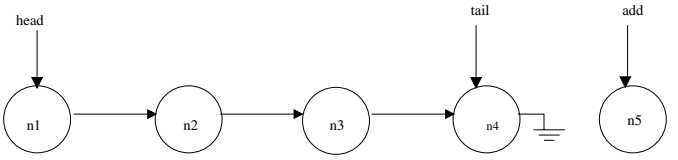


Fig. 3. A linked list with four nodes.

Table 1. List of the heap formulas that can appear in *guard* and *update* parts of a heap transition formula (id_1 and id_2 are heap variables and f is a selector field)

Guard Formulas		Update Formulas	
$id_1 = id_2$	$id_1.f = id_2$	$id'_1 = id_2$	$id'_1 = id_2.f$
$id_1.f = id_2.f$	$id_1 \not= id_2$	$id'_1.f = id_2$	$id'_1.f = id_2.f$
$id_1.f \not= id_2$	$id_1.f \not= id_2.f$	$id'_1 = null$	$id'_1.f = null$
$id_1 = null$	$id_1.f = null$	$id'_1 = new$	$id'_1.f = new$

variables and the heap using a list of *shape graphs*. A shape graph sg is a tuple (N, SR, PT) where

- N is the set of nodes in the shape graph where each node corresponds to a heap cell,
- $SR : Sel \rightarrow \mathcal{P}(N \times (N \cup \{null\}))$ is a function that maps each selector field $f \in Sel$ to a set of tuples R_f such that $\forall (m_1, m_2) \in R_f$ s.t. $m_1 \in N$ and $m_2 \in N \cup \{null\}$, $m_1.f = m_2$, and
- $PT : Var \rightarrow N \cup \{null, undef\}$ is a function that maps each heap variable either to a heap cell, to $null$, or to $undef$ which means uninitialized.

For example, let $Var = \{head, tail, add\}$, and, let $Sel = \{next\}$, then, the state of the heap for the linked list shown in Figure 2 is represented by the following shape graph:

$$N = \{n1, n2, n3, n4\},\ PT(head) = n1,\ PT(tail) = n4,\ PT(add) = n4,$$
$$SR(next) = \{(n1, n2), (n2, n3), (n3, n4), (n4, null)\}$$

We include a node in a shape graph only if it is reachable from a heap variable, i.e., $\forall n \in N, \exists (n_1, n_2, ..., n_k), n_k = n \wedge (\exists v \in Var, PT(v) = n_1) \wedge \forall 1 \leq i < k, (n_i, n_{i+1}) \in SR$.

A heap transition formula, r, can be formulated as $r = r_{guard} \wedge r_{update}$ where r_{guard} has only current state variables and r_{update} has both current state and next state variables. For example, for the formula $a = b.next \wedge a'.next = b$, r_{guard} part is the formula $a = b.next$ and the r_{update} part is the formula $a'.next = b$. Table 1 shows the possible statements that may appear in r_{guard} and r_{update} parts of a heap transition formula r, respectively.

Given a set of shape graphs s and a heap transition formula r we compute the post-condition of s with respect to r as follows:

$$\textsc{post}(s, r) = \bigvee_{sg \in s} \textsc{post}(sg, r)$$

where $\text{POST}(sg, r)$ updates the representation of the heap state (that is $sg.N$, $sg.SR$, and $sg.PT$) according to the assignment part r_{update} of transition relation r if the heap state sg satisfies the enabling condition r_{guard}.

For instance, given the shape graph shown in Figure 2 and the heap transition formula $r \equiv add' = new$, the post-condition algorithm will output the shape graph given in Figure 3. Note that the *new* keyword generates a new node in the heap.

4.1 Encoding Shape Graphs with BDDs

We encode shape graphs symbolically using BDDs. We represent the nodes in the shape graph with minterms on a set of boolean variables (conjunctions of boolean variables or their negations). For example, the nodes of shape graph in Fig. 2 can be represented using two boolean variables b_1, b_2 where the minterm $\neg b_1 \wedge \neg b_2$ encodes node $n1$, $b_1 \wedge \neg b_2$ encodes node $n2$, $\neg b_1 \wedge b_2$ encodes node $n3$, $b_1 \wedge b_2$ encodes node $n4$. Then, function PT corresponds to a function from heap variables to boolean logic formulas on boolean variables b_1 and b_2. For example, $PT(head)$ is the boolean logic formula $\neg b_1 \wedge \neg b_2$ for the shape graph in Fig. 2 based on the encoding given above.

Each selector defines a binary relation on N, the set of nodes in the shape graph. To encode binary relations defined by the selectors, we duplicate the boolean variables used to encode the nodes in the heap. For example for the above example, we add two new boolean variables b'_1 and b'_2. Now, the binary relation on N defined by the selector *next* can be represented as a boolean logic formula on variables b_1, b_2, b'_1, and b'_2. The formula that corresponds to the shape graph in Fig. 2 based on the encodings of the nodes given above is:

$$\neg b_1 \wedge \neg b_2 \wedge b'_1 \wedge \neg b'_2 \;\vee\; b_1 \wedge \neg b_2 \wedge \neg b'_1 \wedge b'_2 \;\vee\; \neg b_1 \wedge b_2 \wedge b'_1 \wedge b'_2$$

This type of encoding is essentially the same encoding used in BDD based model checking to encode transition relations. Note that *null* corresponds to the boolean value $false$. Boolean value $true$ on the other hand is used to encode the configurations where the selector for a node is not initialized.

To keep the size of the BDD encoding small we introduce boolean variables only when they are needed. In our encoding, a shape graph with the set of nodes N will be encoded with $2 \times \log_2(|N|)$ boolean variables. Since new nodes can be added to the heap using the *new* keyword, we introduce new boolean variables to the encoding dynamically if necessary.

4.2 Truncated Fixpoints Computations

Using composite symbolic representation, with shape graphs as a symbolic representation for the heap variables, we can compute the reachable states of a system from the initial set of states up to a certain execution step. Each iteration of the least fixpoint computation for the set of reachable states will give a lower bound

for the reachable states RS. Hence, if we truncate the fixpoint computation after a finite number of iterations we will have a lower bound $RS^- \subseteq RS$ for the reachable states. Our verification tool has a flag which can be set to determine the bound on number of fixpoint iterations. If the obtained result is not precise enough to find an error, the precision can be improved by running more fixpoint iterations. This way we can detect errors up to a certain number of execution steps. However, this kind of bounded verification is not sound, i.e., if there are no states violating the invariants in RS^- this does not mean that there are no errors in the system.

5 Approximations: Summarization and Widening

The heap state representation that we described in Section 4 is a precise representation, i.e. each node in N corresponds to a concrete heap cell. One implication of using a precise representation is that the shape graphs may grow arbitrarily large. To guarantee the termination of the fixpoint computation for the set of reachable states we need to introduce some abstraction. We use the notion of *summary* nodes [CWZ90,SRW98,DRS00] to map more than one concrete heap cell into an abstract node in the shape graph. In this section, we will focus on singly linked lists, i.e., we assume that the selector field set Sel has only one element f.

We define an abstraction function $M : N \rightarrow \hat{N}$ that maps the nodes in a concrete shape graph $sg = (N, SR, PT)$ to the nodes of an abstract shape graph $\hat{sg} = (\hat{N}, \hat{SR}, \hat{PT}, SM)$. Since we assume that there is a single selector field, we encode the selector relation as $SR \subseteq N \times (N \cup \{null\})$. We define the set of summary nodes SM as the abstract nodes which represent more than one concrete node:

$$SM = \{\hat{n} \mid \hat{n} \in \hat{N} \wedge \exists n_1, n_2 \in N, n_1 \neq n_2 \wedge M(n_1) = \hat{n} \wedge M(n_2) = \hat{n}\}$$

We impose the following constraints on function M:

1. $\forall \hat{n_i} \in SM,\ \forall n,\ M(n) = \hat{n_i}, \forall v \in Var, PT(v) \neq n,$
2. $\forall \hat{n_i} \in SM,\ \forall n,\ M(n) = \hat{n_i}, \neg(\exists s, t,\ (s, n) \in SR \wedge (t, n) \in SR, s \neq t),$
3. $\forall \hat{n_i} \in SM,\ \forall m,\ M(m) = \hat{n_i},\ \exists n,\ M(n) = \hat{n_i} \wedge ((m, n) \in SR \vee (n, m) \in SR)$

Constraint (1) states that no heap variable points to any of the concrete nodes that are mapped to a summary node. Constraint (2) states that there cannot be more than two heap pointers to any of the concrete nodes that are mapped to a summary node. Constraint (3) states that all the concrete nodes mapped to the same summary node are connected. Note that, based on these constraints, each summary node represents a chain of nodes such that no heap variable points to any of the nodes in the chain and no node in the chain is pointed by more than one heap pointer.

For instance, the set of abstract nodes, $\hat{N}$, of the abstract shape graph for the linked list in Figure 2 is $\{\hat{n_1}, \hat{n_2}, \hat{n_3}\}$ where $M(n_1) = \hat{n_1}$, $M(n_2) = \hat{n_2}$,

$M(n_3) = \hat{n_2}$, and $M(n_4) = \hat{n_3}$. The set of summary nodes for this abstract shape graph is $SM = \{\hat{n_2}\}$.

The abstract selector relation $\hat{SR}$ is defined as follows:

$$\begin{aligned}\hat{SR}(f) = \{(\hat{n_1}, \hat{n_2}) \mid \hat{n_1} \in \hat{N} \wedge \hat{n_2} \in \hat{N} \cup \{null\} \ \wedge \\ \exists n_1, n_2,\ M(n_1) = \hat{n_1} \wedge M(n_2) = \hat{n_2} \wedge \\ (M(n_1) \ /\!\!= M(n_2) \vee M(n_1) \ /\!\!\in SM \vee M(n_2) \ /\!\!\in SM) \wedge (n_1, n_2) \in SR \ \}\end{aligned}$$

For instance, for the abstract shape graph that corresponds to the linked list in Figure 2, $\hat{SR} = \{(\hat{n_1}, \hat{n_2}), (\hat{n_2}, \hat{n_3}), (\hat{n_3}, null)\}$. Note that, the internal edges of the chain represented by a summary node is not represented in $\hat{SR}$.

$\hat{PT}$ is defined as follows:

$$\hat{PT}(v) = \begin{cases} undef & PT(v) = undef \\ null & PT(v) = null \\ \hat{n} & M(m) = \hat{n} \wedge PT(v) = m \end{cases}$$

For instance, for the abstract shape graph that corresponds to the linked list given in Figure 2, $\hat{PT}(head) = \hat{n_1}, \hat{PT}(tail) = \hat{PT}(add) = \hat{n_3}$.

Abstraction of a concrete shape graph is an abstract shape graph with minimum number of nodes which does not violate any of the conditions discussed above. Note that, this abstraction causes a loss of precision by mapping chains of arbitrary lengths to summary nodes. Concretization of an abstract shape graph would generate an infinite set of concrete shape graphs where each summary node is mapped to an infinite number of chains. Based on the abstraction defined above, the set of all possible abstract shape graphs for singly linked lists with a given number of heap variables is finite [YKB02]. (Note that, we are only representing nodes reachable from heap variables.) Since the set of all possible abstract shape graphs are finite, we can guarantee termination for the fixpoint computations using the abstract shape graphs. However, we lose the precision that existed in the concrete representation. Below, we propose a technique that: 1) guarantees termination and 2) yields a more precise representation than the abstract shape graphs that only use summary nodes.

We achieve the above features by associating each summary node s_i in an abstract shape graph sg with a summary count $count_{s_i}$ that keeps the number of concrete nodes that s_i represents. As we discussed earlier a composite symbolic representation is in the form $S \equiv \bigvee_{i=1}^{n} s_{i_{bool}} \wedge s_{i_{int}} \wedge s_{i_{heap}}$ where each composite atom $s_{i_{bool}} \wedge s_{i_{int}} \wedge s_{i_{heap}}$ consists of a boolean logic formula $s_{i_{bool}}$ (encoded as a BDD), a Presburger arithmetic formula $s_{i_{int}}$ (encoded as a set of polyhedra), and a set of shape graphs $s_{i_{heap}}$. We encode the valuations for the summary count variables for the shape graphs in $s_{i_{heap}}$ in the Presburger arithmetic formula $s_{i_{int}}$.

Initially, when a summary node is created, its summary count is assigned to a value equal to the number of heap cells it summarizes. Whenever we update a shape graph during the post-condition computation based on an update formula given in Table 1 we need to update the summary counts accordingly. We achieve this by generating an integer formula which specifies how the summary counts

```
SUMMARIZE(shape graph sg = (N̂, ŜR, P̂T, SM))
  Remove all the nodes that are not reachable from a heap variable
  Initialize constant and coef arrays to 0, marked array to false and removed to ∅
  for each heap variable v do  marked[P̂T(v)] ← true;
  for each node n do
    if ∃s, t, s ≠ ∧ (s, n) ∈ ŜR ∧ (t, n) ∈ ŜR then
      marked[n] ← true;
  for each heap variable v such that P̂T(v) ∉ {undef, null} do
    s ← P̂T(v);
    while ∃(s, t) ∈ ŜR ∧ t ≠ null do
      if marked[s] ∨ marked[t] then
        s ← t;
      else
        if s ∉ SM ∧ t ∈ SM then
          N̂ ← N̂ \ {s};
          ŜR ← ŜR \ ({(s, t)} ∪ {(m, s)|(m, s) ∈ ŜR}) ∪ {(m, t)|(m, s) ∈ ŜR};
          constant[t] ← constant[t] + 1;
          s ← t;
        else
          N̂ ← N̂ \ {t};
          SM ← SM ∪ {s};
          ŜR ← ŜR \ ({(s, t)} ∪ {(t, m)|(t, m) ∈ ŜR}) ∪ {(s, m)|(t, m) ∈ ŜR};
          if s ∈ SM ∧ t ∈ SM then
            SM ← SM \ {t};
            coef[s, t] ← 1;
            removed ← removed ∪ {t};
          else if s ∈ SM ∧ t ∉ SM then
            constant[s] ← constant[s] + 1;
          else // s ∉ SM ∧ t ∉ SM
            constant[s] ← 2;
  sg.F ← ⋀_{t∈removed} count'_t = 0 ∧
    ⋀_{s∈SM} count'_s = count_s + ∑_{t∈removed} coef[s, t] × count_t + constant[s];
```

Fig. 4. Summarization algorithm

should be updated for each generated shape graph during the post-condition computation.

Figure 4 shows the summarization algorithm we use. SUMMARIZE algorithm is called whenever an update to a shape graph may create a chance for summarization. The algorithm first does the garbage collection; it removes the nodes that are not reachable by any heap variable. Then it merges nodes and replaces them with summary nodes respecting the constraints (1), (2), and (3) imposed on M. While updating the summary nodes, SUMMARIZE algorithm also constructs an arithmetic constraint ($sg.F$) which reflects the effect of the updates on the summary counts.

SETNULL algorithm is used to implement the effect of update formulas of type $x' = null$ on shape graphs. Given a shape graph sg and a heap variable x,

POST(composite atom $s \equiv s_{int} \wedge s_{bool} \wedge s_{heap}$, composite transition formula $r \equiv r_{int} \wedge r_{bool} \wedge r_{heap}$): composite formula
  $result$: composite formula
  $heapf$: list of shape graphs
  $intf$: integer formula
  $boolf$: boolean formula
  $result \leftarrow$ false;
  $boolf \leftarrow$ POST(s_{bool},r_{bool});
  $heapf \leftarrow$ POST(s_{heap},r_{heap});
  for each shape graph h_{sg} in $heapf$ **do**
    $intf \leftarrow$ POST($s_{int}, r_{int} \wedge h_{sg}.F$);
    $result \leftarrow result \vee (intf \wedge boolf \wedge h_{sg})$
  return $result$;

Fig. 5. An algorithm for computing the post-condition of a composite atom state formula s with respect to a composite atom transition formula r

SETNULL algorithm first sets the heap variable x to null. Since setting x to null may create a chance for summarization it calls SUMMARIZE algorithm:

SETNULL(shape graph sg, heap variable x)
  $sg.\hat{PT}(x) \leftarrow null$;
  SUMMARIZE(sg);

Given a shape graph sg and an update formula $x' = y.f$, SETTONEXT algorithm (Figure 6) makes heap variable x point to the node that is pointed by $y.f$. It first calls SETNULL algorithm to set heap variable x to null. Summarization is handled inside SETNULL algorithm. Then SETTONEXT algorithm performs a materialization operation if m, where $y.f = m$, is a summary node. In this case two different shape graphs, sg_1 and sg_2, are generated. Both sg_1 and sg_2 are initialized with a copy of sg. sg_1 corresponds to the case that m abstracts exactly two concrete nodes ($count_m = 2$). In sg_1, m is removed ($count'_m = 0$) and it is replaced by two nodes, v and w, each representing a concrete node. At the end sg_1 satisfies $x = y.f = v \wedge v.f = w$. sg_2 corresponds to the case that m abstracts more than 2 concrete nodes ($count_m > 2$). In sg_2, m is kept as a summary node which abstracts one less concrete nodes ($count'_m = count_m - 1$). Only one new node, v, that represents a concrete node is introduced in sg_2. At the end sg_2 satisfies $x = y.f = v \wedge v.f = m$. On the other hand, if m is not a summary node then the algorithm simply sets x to $y.f$ and the resulting shape graph satisfies $x = y.f = m$. The existential quantification and renaming used in combining $sg_1.F$ and $sg_2.F$ with $sg.F$ is required to compose the constraints generated by the summarization and the materialization operations. Since the arithmetic constraint manipulator used in the Composite Symbolic Library can handle Presburger arithmetic we are able to use quantification in the generated formulas.

We compute the post-condition for shape graphs using algorithms SETNULL, SETTONEXT, and similar algorithms for other update formulas as follows:

```
POST(shape graph sg, heap transition formula r): set of shape graphs
    switch (r)
        case (x' = null):
            SETNULL(sg,x);
            return {sg};
        case (x' = y.f):
            return SETTONEXT(sg,x,y);
        ...
```

Using the post condition computation given above we implement the post condition computation for a composite atom as shown in Figure 5. Then, the post condition computation of the overall transition relation is computed as:

```
POST(composite representation S ≡ ⋁_i s_i, composite transition formula R ≡
⋁_j r_j): composite representation
    result: composite representation
    result ← false;
    for each composite atom s_i in S do
        for each disjunct r_j in R do
            result ← result ∨ POST(s_i,r_j);
    return result;
```

where the composite transition formula is stored in a disjunctive form (similar to the composite representation of states).

Note that, the summary counts for summary nodes can increase without a bound. Actually, our input language also allows unbounded integer variables. To force the fixpoint computation for reachable states to converge we use the *widening* technique [CC77]. Note that, the shape graphs without summary counts, and the boolean and enumerated variables are bounded. Hence, the only cause for nontermination is in the integer symbolic representation. For the integer symbolic representation we use the widening operator defined in [BGP99] for Presburger arithmetic constraints by generalizing the convex polyhedra widening operator in [CH78]. The basic idea is to find pairs of polyhedra p and q such that $p \subseteq q$ and set $p \bigtriangledown q$ to conjunction of constraints in p which are also satisfied by q. Intuitively, if a constraint of p is not satisfied by q this means that the iterates are increasing in that direction. By removing that constraint we extend the iterates in the direction of growth as much as possible without violating other constraints.

6 Experimental Results

We experimented our technique by verifying the safety properties of three different concurrent data structure specifications: single lock queue, stack and two lock queue [Yah01]. Table 3 shows the verified safety properties for each data structure specification. All of the examples use an integer variable to keep the number of items stored in the data structure. We verified each instance with

```
SETTONEXT(shape graph sg, heap variable x, heap variable y): set of shape graphs
  sg ← SETNULL(sg, x);
  assert (∃(n, m) ∈ sg.ŜR, sg.P̂T(y) = n);
  if m ∈ sg.SM then // materialization operation
    sg1 ← getCopy(sg);
    sg1.N̂ ← sg.N̂ \ {m} ∪ {v, w};
    sg1.SM ← sg.SM \ {m};
    sg1.F ← count_m = 2 ∧ count'_m = 0;
    sg1.F ← ∃count''_m(sg.F[count'_m ↦ count''_m] ∧ sg1.F[count_m ↦ count''_m]);
    sg1.ŜR ← sg.ŜR \ ({(t, m)|(t, m) ∈ sg.ŜR} ∪ {(m, t)|(m, t) ∈ sg.ŜR})
              ∪{(t, v)|(t, m) ∈ sg.ŜR} ∪ {(w, t)|(m, t) ∈ sg.ŜR} ∪ {(v, w}};
    sg1.P̂T(x) ← v ;

    sg2 ← getCopy(sg);
    sg2.N̂ ← sg2.N̂ ∪ {v};
    sg2.F ← count_m > 2 ∧ count'_m = count_m − 1;
    sg2.F ← ∃count''_m(sg.F[count'_m ↦ count''_m] ∧ sg2.F[count_m ↦ count''_m]);
    sg2.ŜR ← sg.ŜR \ {(t, m)|(t, m) ∈ sg.ŜR} ∪ {(t, v)|(t, m) ∈ sg.ŜR} ∪ {(v, m)};
    sg2.P̂T(x) ← v;
    return {sg1, sg2};
  else // m ∉ sg.SM
    sg1 ← getCopy(sg);
    sg1.P̂T(x) ← m;
    return {sg1};
```

Fig. 6. An algorithm for computing the post-condition of a heap state formula represented by a shape graph sg with respect to update formula $x' = y.f$

Table 2. Verification timing results (in seconds) for the concurrent single lock queue, stack, and two lock queue for the safety properties given in Table 3. CT and VT denote construction time and verification time in seconds, respectively. HC and IC refer to two different specifications for each problem in which the control flow is based only on heap formulas (using tests such as $head \neq null$) or both heap and integer formulas (using tests such as $numItems > 0$), respectively. ↑ denotes that the verifier ran out of memory.

Number of Processes	Queue				Stack				TwoLockQueue			
	HC		IC		HC		IC		HC		IC	
	CT	VT	CT	VT	CT	VT	CT	VT	CT	VT	CT	VT
1Prod-1Cons	0.01	10.19	0.01	12.95	0.01	4.57	0.02	5.21	0.01	60.5	0.04	58.13
2Prod-2Cons	0.03	15.74	0.09	21.64	0.02	6.73	0.04	8.24	0.69	88.26	0.59	122.47
4Prod-4Cons	9.07	31.55	19.38	46.5	5.56	12.71	4.98	15.11	↑	↑	↑	↑
1Prod-PCons	0.01	12.85	0.02	13.62	0.01	5.61	0.02	5.73	↑	↑	↑	↑
PProd-1Cons	0.01	18.24	0.02	19.43	0.01	6.48	0.01	6.82	↑	↑	↑	↑

varying number of producer and consumer process types. Table 2 shows the verification timing results (in seconds) for these examples. $PProd$ ($PCons$) means

Table 3. Safety properties of the concurrent data structure specifications.

Specification	Verified Invariants
Single Lock Queue	$(head = null) \Leftrightarrow numItems = 0$ $(head \neq null) \Leftrightarrow numItems > 0$ $(head = tail \wedge head \neq null) \Leftrightarrow numItems = 1$ $(head \neq tail) \Leftrightarrow numItems \geq 2$
Stack	$numItems = 0 \Leftrightarrow top = null$ $numItems > 0 \Leftrightarrow top \neq null$ $numItems = 2 \Rightarrow top.next \neq null$
Two Lock Queue	$numItems > 1 \Rightarrow head \neq tail$ $numItems > 2 \Rightarrow head.next \neq tail$

that the specification is verified for arbitrary number of producer (consumer) processes. We used counting abstraction [Del00] to verify parameterized cases.

Results show that the verification time is scalable with the increasing number of processes. Most of the time the verification of the specifications for arbitrary number of producer or consumer processes are faster than that of the cases where there are fixed number of processes. This is possibly due to the fact that counting abstraction, in a way, implements a symmetry reduction by abstracting away the information about individual processes. The results also show that the verification time for the case where the emptiness check is expressed using an integer formula is greater than the verification time for the case where the emptiness check is expressed using a heap formula.

7 Conclusions

The approach presented in this paper combines previous work on shape analysis, infinite state verification using arithmetic constraints and symbolic model checking to obtain a powerful verification technique. Using Composite Symbolic Library [YKTB01] we were able to integrate shape analysis to the composite representation framework. We think that we will be able to generalize the approach presented here to verification of liveness properties and backward fixpoint computations in the future. Extending the presented approach to heap objects with multiple selector fields (i.e., doubly linked lists) is another future research direction.

References

[AHH96] R. Alur, T. A. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996.

[BGP99] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, July 1999.

[BYK01] T. Bultan and T. Yavuz-Kahveci. Action Language Verifier. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, 2001.

[CC77] 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 the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming*, pages 84–97, 1978.

[CUD] CUDD: CU decision diagram package. `http://vlsi.colorado.edu/~fabio/CUDD/`

[CWZ90] D.R. Chase, M. Wegman, and F.K. Zadeck. Analysis of pointers and structures. In *ACM SIGPLAN Conference on Program Language Design and Implementation*, 1990.

[Del00] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68, 2000.

[DRS00] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In Jens Palsberg, editor, *7th International Static Analysis Symposium*, Lecture Notes in Computer Science. Springer, 200.

[HRP94] N. Halbwachs, P. Raymond, and Y. Proy. Verification of linear hybrid systems by means of convex approximations. In B. LeCharlier, editor, *Proceedings of International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1994.

[LARSW00] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing And Analysis*, 2000.

[LAS00] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analysis. In Jens Palsberg, editor, *7th International Static Analysis Symposium*, Lecture Notes in Computer Science. Springer, 200.

[Ome] The Omega project. `http://www.cs.umd.edu/projects/omega/`

[SRW98] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.

[Yah01] E. Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. In *ACM Symposium on Principles of Programming Languages*, 2001.

[YKB02] T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. Technical Report 2002-19, University of California, Santa Barbara, 2002.

[YKTB01] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representation. In *Proceedings of the Seventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, 2001.

Backward Type Inference Generalises Type Checking

Lunjin Lu[1] and Andy King[2]

[1] Oakland University, MI 48309, USA.
[2] University of Kent at Canterbury, CT2 7NF, UK.

Abstract. This paper presents a backward type analysis for logic programs. Given type signatures for a collection of selected predicates such as builtin or library predicates, the analysis infers type signatures for other predicates such that the execution of any query satisfying the inferred type signatures will not violate the type signatures for the selected predicates. Thus, the backward type analysis generalises type checking in which the programmer manually specifies type signatures for all predicates that are checked for consistency by a type checker.

1 Introduction

This paper focuses on the inference of type signatures for predicates in a logic program that ensure the execution of the program with a query satisfying the inferred type signatures will be free from type errors. This generalises type checking in which the programmer declares type signatures for all predicates in the program and a type checker verifies if the program is well-typed with respect to these type signatures, that is, these type signatures are consistent with the operational semantics of the program. For instance, for the quicksort program in Prolog, a type checker would require the programmer to declare type signatures for $quicksort/2$, $partition/4$ and $append/3$ and then check if the program is well-typed with respect to these and the type signatures for builtin predicates $> /2$ and $=< /2$ stipulated in the Prolog language manual. In contrast, type signature inference will infer that if $quicksort/2$ is called with a list of numbers as the first argument then the execution of the program will not violate the type signatures of $> /2$ and $=< /2$; the programmer need not declare types for $quicksort/2, partition/4$ nor $append/3$.

Type analyses traditionally propagate type information in the direction of program execution and cannot infer type signatures (see section 7 for a short survey). One notable exception is the backward analysis framework of [15] that can, in principle, be instantiated with a type domain. However, the framework propagates information backwards using the (intuitionistic) logical implication operator and this requires domains to be closed under Heyting completion which is a strong requirement by any standard. The backward type analysis proposed in this paper is based on a novel abstract semantics (that will be presented in a companion paper) that relaxes this requirement. The backward type analysis is obtained by specialising the new abstract semantics with domain operators for type inference and in particular type inference using an abstract domain [18] which is particularly precise since it is disjunctive and supports non-deterministic regular types. The resulting analysis is the first analysis that infers type signatures for a logic program. It gives the programmer the flexibility not to declare and maintain type signatures for

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 85–101, 2002.
© Springer-Verlag Berlin Heidelberg 2002

predicates that are subject to frequent modifications during program development. In the extreme situation, the programmer may choose to leave unspecified type signatures for all user-defined predicates and let the analyser to infer type signatures from builtin and library predicates. One application of the new analysis is automatic program documentation; type signatures provide valuable information for both program development and maintenance. Another application is in bug detection. The inferred type signature for a predicate can be compared with that intended by the programmer and any discrepancy indicates the possible existence of bugs.

The remainder of the paper is organised as follows. Section 2 motivates the analysis with a worked example. Section 3 introduces the fundamental concepts used in the paper and outlines the abstract semantics on which the backward type analysis is based. Sections 4 and 5 present the abstract domain and the abstract operations for the backward type analysis. A prototype implementation and analysis examples are presented in section 6 and related work is discussed in section 7. Section 8 concludes. Proofs are omitted due to space limitations.

2 Worked Example

This section informally presents a backward type analysis that deduces typing properties of the call which, if satisfied, guarantee that the execution of the call can never encounter a type error. Consider the following partition program.

$pt(xs, x, ys, zs)$:- ⓪ $xs = [y|xs']$, $ys = [y|ys']$,① $y =< x$,② $pt(xs', x, ys', zs)$.③
$pt(xs, x, ys, zs)$:- ④ $xs = [z|xs']$, $zs = [z|zs']$, ⑤ $z > x$, ⑥ $pt(xs', x, ys, zs')$.⑦
$pt(xs, x, ys, zs)$:- ⑧ $xs = [\,]$, $ys = [\,]$, $zs = [\,]$.⑨

The program is in a normalised form in which the arguments of atoms are distinct variables. Textual points of interest in the program are numbered.

The analysis presupposes the existence of type assertions that are type constraints which must be respected by program execution. A type constraint is a disjunction of primitive type constraints. A primitive type constraint is a conjunction of atomic type constraints of the form $x \in R$ where x is a variable and R a type that denotes a set of terms closed under instantiation; x is said to be of type R if $x \in R$ (see section 4). The type assertions are given by a map $\mathcal{A}$ from atoms to type constraints. The type assertions for the partition program are as follows where type num denotes the set of numbers.

$$\mathcal{A}(x > y) = x \in num \wedge y \in num$$
$$\mathcal{A}(x =< y) = x \in num \wedge y \in num$$
$$\mathcal{A}(pt(xs, x, ys, zs)) = true$$

The above type assertions stipulate that the only program states in which $x > y$ or $x =< y$ can be called are those in which both x and y are numbers. For the partition program, the only non-trivial type assertions arise from builtins. This would change if the programmer introduced type assertions for the purposes of verification.

The analysis also requires a set of type rules that gives the denotations of types. For the partition program, type rules are $list(\beta) \rightarrowtriangle [\,]$ and $list(\beta) \rightarrowtriangle [\beta|list(\beta)]$ that define polymorphic lists, According to these two type rules, $[1, 2]$ is of type $list(num)$ while $[1|2]$ is not.

The analysis is formulated in terms of abstract interpretation that iteratively refines and updates a function until stability is reached. The function maps a demand $\langle p(\boldsymbol{x}), \phi_R \rangle$ to a type constraint ϕ_L where $\boldsymbol{x}$ denotes a tuple of distinct variables and ϕ_R a type constraint. The guarantee is that if the execution of $p(\boldsymbol{x})$ begins in a state satisfying ϕ_L, then the execution will respect type assertions in $\mathcal{A}$ and, if it succeeds, terminate in a state satisfying ϕ_R. ϕ_R is called a type post-condition and ϕ_L a type pre-condition for $\langle p(\boldsymbol{x}), \phi_R \rangle$. The type pre-condition for the demand $\langle p(\boldsymbol{x}), true \rangle$ specifies the type signature for $p(\boldsymbol{x})$. Note that there may be more than one demand for a predicate.

The analysis is performed by computing a greatest fixpoint (gfp). One starts with assuming no call causes a type error and then checks this assumption by reasoning backwards over all clauses. If a type assertion is violated, type pre-conditions for demands are strengthened (made smaller), and the whole process is repeated until the assumptions turn out to be valid (the gfp is reached). A series $D_i : i \geq 0$ of iterates is computed. Each iterate D_i that is computed corresponds to a function that maps a demand to a type pre-condition. The iterate contains only those demands whose type pre-conditions are required for the analysis to proceed. If a demand is not in the iterate, its type pre-condition is assumed to be $true$.

The process is exemplified by computing a type pre-condition for a demand $\langle pt(xs, x, ys, zs), true \rangle$. The initial iterate is

$$D_0 = \{\langle pt(xs, x, ys, zs), true \rangle \mapsto true\}$$

Each clause for $pt/4$ is used to compute the type pre-condition for the demand $\langle pt(xs, x, ys, zs), true \rangle$ in D_1. Illustrated below are steps in reasoning backwards over (from right to left) the body of the first clause for $pt/4$. The type constraint for each program point in the clause is listed. The comment following the program point explains the action taken to compute the type constraint for the next program point to the left. Types num and $list(num)$ are abbreviated as n and $l(n)$ respectively.

$$\begin{array}{rl}
true & ③ \\
true & ② \text{ builtin predicate} \\
(x \in n \wedge y \in n) & ① \text{ abstract unification} \\
(x \in n \wedge y \in n) \vee (xs \in l(n) \wedge x \in n) \vee (x \in n \wedge ys \in l(n)) & ⓪
\end{array}$$

The demand $\langle pt(xs', x, ys', zs), true \rangle$ is a variant of $\langle pt(xs, x, ys, zs), true \rangle$ which is in D_0. Thus, the type constraint for point ② is $true$ which is obtained by applying D_0 and renaming. The type constraint $(x \in n \wedge y \in n)$ at point ① results from the type assertion for $=<$ in $\mathcal{A}$. The type constraint at point ⓪ is obtained by performing a backward abstract unification. An informal explanation follows. The conjunct $(x \in n \wedge y \in n)$ derives from the fact that a type constraint that holds in the program state before unification also holds in the program state after unification. The conjunct $xs \in l(n) \wedge x \in n$ is derived as follows. Both xs and $[y|xs']$ are of the same type after unification $xs = [y|xs']$. If $xs \in l(n)$ then $[y|xs'] \in l(n)$ which implies $y \in n$. Therefore, if $(xs \in l(n) \wedge x \in n)$ holds at point ⓪ then $(x \in n \wedge y \in n)$ holds at point ①. The conjunct $(x \in n \wedge ys \in l(n))$ is derived similarly. The type constraint at point ⓪ is *universally* quantified with respect to variables $\{y, xs', ys'\}$ to compute a type constraint that is only defined in terms of the variables in the head of the clause.

This gives type constraint $(xs \in l(n) \wedge x \in n) \vee (x \in n \wedge ys \in l(n))$ which is strictly stronger than the type constraint at point ⓪. More generally, universal quantification of a type constraint ϕ obtains a type constraint ϕ' that is at least as strong as ϕ in that a program state θ satisfies ϕ whenever θ satisfies ϕ'. By restricting a type constraint in this way to variables in the head, the type constraints are kept small and manageable. Universal quantification is required because weakening type constraints compromises safety whereas strengthening the type constraints preserves safety.

Processing the second and third clauses for $pt/4$ gives type constraints $(xs \in l(n) \wedge x \in n) \vee (x \in n \wedge zs \in l(n))$ and $true$ respectively. The three type constraints obtained are conjoined to give the new type pre-condition $(xs \in l(n) \wedge x \in n) \vee (x \in n \wedge ys \in l(n) \wedge zs \in l(n))$ for $\langle pt(xs, x, ys, zs), true\rangle$. Conjunction is required since type assertions must be respected no matter which clause is selected. Thus,

$$D_1 = \{\langle pt(xs, x, ys, zs), true\rangle \mapsto (xs \in l(n) \wedge x \in n) \vee (x \in n \wedge ys \in l(n) \wedge zs \in l(n))\}$$

Omitting the details of the computation of D_2, the gfp is reached at D_1 since $D_2 = D_1$. The gfp expresses safe call pattern; it states that $pt/4$ cannot generate a type error if either it is called with a list of numbers as its first argument and a number as its second argument or it is called with a number as its second argument and lists of numbers as its third and fourth arguments. Manual inspection might miss the latter.

3 Preliminaries

We now recall some terminology of logic programming and that of abstract interpretation. The reader is referred to [16] and [6] for more detailed exposition.

3.1 Basic Concepts

Let Σ be a set of function symbols, $\mathcal{V}$ a denumerable set of variables. We assume that Σ contains at least one function symbol of arity 0. $\mathsf{Term}(\Sigma, V)$ denotes the set of terms that can be constructed from Σ and V where $V \subseteq \mathcal{V}$.

The object resulting from renaming a variable x in object o to another variable y is denoted $\rho_{x \mapsto y}(o)$. Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be vectors of variables of the same dimension. $\rho_{\boldsymbol{x} \mapsto \boldsymbol{y}}(o)$ is the result of simultaneously replacing elements of $\boldsymbol{x}$ in o with corresponding elements of $\boldsymbol{y}$. An equation over V is a formula of the form $t_1 = t_2$ with $t_1, t_2 \in \mathsf{Term}(\Sigma, V)$. An equational constraint is a finite set (conjunction) of equations. The set of all equational constraints is denoted as Eqn whereas the set of all idempotent substitutions is denoted Sub. Let $Sub_{fail} = Sub \cup \{fail\}$. Given $E \in Eqn$, $mgu : Eqn \mapsto Sub_{fail}$ returns either a most general unifier for E if E is unifiable or $fail$ otherwise. For brevity, let $mgu(t_1, t_2) \doteq mgu(t_1 = t_2)$. The function composition operator $\circ$ is defined as $f \circ g \doteq \lambda x.f(g(x))$. Sub is ordered by *less general than*, that is, $\theta_1 \leq \theta_2$ iff there exists $\delta \in Sub$ such that $\theta_1 = \delta \circ \theta_2$. The ordering $\leq$ extends to Sub_{fail} by $fail \leq \theta$ for all $\theta \in Sub_{fail}$. The equivalence between substitutions is defined as $\theta_1 \sim \theta_2$ iff $\theta_1 \leq \theta_2$ and $\theta_2 \leq \theta_1$. Furthermore, let $\theta \circ fail \doteq fail$ and $fail \circ \theta \doteq fail$ for any $\theta \in Sub_{fail}$.

Let VI be the set of variables in the program, $dom(\theta)$ the domain of an idempotent substitution θ, and $Var(o)$ the set of variables in the syntactic object o. Existential

quantification $\exists_x : Sub_{fail} \mapsto Sub_{fail}$ for $x \in VI$ is defined as $\exists_x(fail) \doteq fail$ and, for $\theta \in Sub$, $\exists_x(\theta) \doteq \{y \mapsto p(\theta(y)) \mid y \in dom(\theta) \wedge y \neq x\}$ where $\rho = \{x \mapsto z\}$ and $z \notin (VI \cup Var(\theta))$. Thus, $\exists_x(\theta)$ projects out x if $x \in dom(\theta)$ and otherwise renames x to a fresh variable if $x \in Var(\theta)$. Universal quantification $\forall_x : Sub_{fail} \mapsto Sub_{fail}$ for $x \in VI$ is defined as $\forall_x(\theta) \doteq$ (if $x \notin Var(\theta)$ then θ else $fail$) . If $x \notin Var(\theta)$ then θ does not constrain x, hence $\forall_x(\theta) = \theta$. Observe that $\forall_x(\theta) \leq \theta \leq \exists_x(\theta)$.

Let $\boldsymbol{x} = \{x_1, x_2, \cdots, x_n\}$ be a subset of VI. Define $\exists_{\boldsymbol{x}}(\theta) \doteq \exists_{x_1}(\exists_{x_2}(\cdots \exists_{x_n}(\theta)))$ and $\forall_{\boldsymbol{x}}\theta \doteq \forall_{x_1}(\forall_{x_2}(\cdots \forall_{x_n}(\theta)))$. Also, $\bar{\exists}_{\boldsymbol{x}}(\theta) \doteq \exists_{VI \setminus \boldsymbol{x}}(\theta)$ and $\bar{\forall}_{\boldsymbol{x}}(\theta) \doteq \forall_{VI \setminus \boldsymbol{x}}(\theta)$.

Let $\langle C, \sqsubseteq^C, \sqcup^C, \sqcap^C, \top^C, \bot^C \rangle$ be a complete lattice and $S \subseteq C$. S is a Moore family iff $\top^C \in S$ and $(s_1 \sqcap^C s_2) \in S$ for any $s_1, s_2 \in S$. Let $\langle D, \sqsubseteq^D \rangle$ be a poset. A function $\gamma : D \mapsto C$ is a concretization function iff γ is monotone and $\gamma(D)$ is a Moore family. A concretization function from D to C induces a Galois connection between D and C [6]. The Kleene closure of a set S is denoted as S^*.

3.2 Abstract Semantics

The backward type analysis is based on a novel abstract semantics for backward analysis of logic programs that is sketched below so that the paper is self-contained. The abstract semantics is a (lower) approximation to a collecting semantics that maps a call $p(\boldsymbol{x})$ and a set Θ of substitutions into a set Ξ of substitutions such that, for any $\xi \in \Xi$, if δ is a computed answer for $\xi(p(\boldsymbol{x}))$ then $\delta \circ \xi \in \Theta$, i.e., $\{\Xi\}p(\boldsymbol{x})\{\Theta\}$ is a valid partial correctness formula. Note that $\Xi = \emptyset$ is valid solution. In a more precise solution, Ξ contains more substitutions while maintaining partial correctness. The collecting semantics is defined on the concrete domain $\langle \wp(Sub), \subseteq \rangle$ and in terms of six operations: $\cap : \wp(Sub) \times \wp(Sub) \mapsto \wp(Sub)$, $uf_{bw} : Eqn \times \wp(Sub) \mapsto \wp(Sub)$, $ex_{bw} : VI^* \times \wp(Sub) \times \wp(Sub) \mapsto \wp(Sub)$, $\exists_x : \wp(Sub) \mapsto \wp(Sub)$ for $x \in VI$, $\forall_x : \wp(Sub) \mapsto \wp(Sub)$ for $x \in VI$ and $\rho_{\boldsymbol{x} \mapsto \boldsymbol{y}} : \wp(Sub) \mapsto \wp(Sub)$ where $\cap$ is the set intersection operation and the others are defined as

$$
\begin{aligned}
uf_{bw}(E, \Theta) &\doteq \{\xi \mid mgu(\xi(E)) \circ \xi \in \Theta\} \\
ex_{bw}(\boldsymbol{x}, \Omega, \Theta) &\doteq \left\{ \xi \,\middle|\, \begin{array}{l} (\bar{\exists}_{\boldsymbol{x}}(\xi) \in \Omega) \\ \wedge\ \forall \delta \in Sub.(\exists \theta \in \Theta.(\delta \circ \bar{\exists}_{\boldsymbol{x}}(\xi) \sim \bar{\exists}_{\boldsymbol{x}}(\theta)) \rightarrow (\delta \circ \xi \in \Theta)) \end{array} \right\} \\
\exists_x(\Theta) &\doteq \{\exists_x(\theta) \mid \theta \in \Theta\} \\
\forall_x(\Theta) &\doteq \{\theta \in \Theta \mid \forall_x(\theta) \sim \theta\} \\
\rho_{\boldsymbol{x} \mapsto \boldsymbol{y}}(\Theta) &\doteq \{\rho_{\boldsymbol{x} \mapsto \boldsymbol{y}}(\theta) \mid \theta \in \Theta\}
\end{aligned}
$$

The collecting semantics guarantees that $\{\Omega\}p(\boldsymbol{x})\{\bar{\exists}_{\boldsymbol{x}}(\Theta)\}$ is a valid partial correctness formula for some $p(\boldsymbol{x})$ before $ex_{bw}(\boldsymbol{x}, \Omega, \Theta)$ is called to extend Ω to obtain a set Ξ of substitutions such that $\{\Xi\}p(\boldsymbol{x})\{\Theta\}$ is also valid. This is guaranteed by requiring, for each $\xi \in \Xi$, (1) $\bar{\exists}_{\boldsymbol{x}}(\xi) \in \Omega$ and (2) if δ is a computed answer for $(\bar{\exists}_{\boldsymbol{x}}(\xi))(p(\boldsymbol{x}))$ then $\delta \circ \xi \in \Theta$.

The abstract semantics is parameterised by an abstract domain $\langle Z, \sqsubseteq^Z \rangle$ and six operations: $\sqcap^Z : Z \times Z \mapsto Z$, $uf^Z_{bw} : Eqn \times Z \mapsto Z$, $ex^Z_{bw} : VI^* \times Z \times Z \mapsto Z$, $\exists^Z_x : Z \mapsto Z$, $\forall^Z_x : Z \mapsto Z$ and $\rho^Z_{\boldsymbol{x} \mapsto \boldsymbol{y}} : Z \mapsto Z$ with $\sqcap^Z$ being the greatest lower bound

operation on Z. The correctness of a backward analysis is guaranteed by requiring that there be a concretization function $\gamma^Z : Z \nrightarrow \wp(Sub)$ and that each operation on Z safely approximates its corresponding operation on $\wp(Sub)$ from below with respect to γ^Z. For instance, $\exists_x^Z : Z \nrightarrow Z$ approximates $\exists_x : \wp(Sub) \nrightarrow \wp(Sub)$ from below iff $\gamma^Z(\exists_x^Z(z)) \subseteq \exists_x(\gamma^Z(z))$ for all $z \in Z$. Note that $\sqcap^Z$ safely approximates $\cap$ from below with respect to γ^Z since γ^Z is a concretization function. Furthermore, if γ^Z is additive, that is, $\gamma^Z(S) = \bigcup_{s \in S} \gamma^Z(s)$ for any $S \subseteq Z$ then it turns out that the precision of the abstract semantics can be improved because the disjunction of two or more pre-conditions for a demand is then a pre-condition for the demand. An abstract domain $\langle Z, \sqsubseteq^Z \rangle$ with an additive γ^Z can be constructed from a pre-order $\langle K, \sqsubseteq^K \rangle$ with a monotone $\gamma^K : K \nrightarrow \wp(Sub)$ satisfying $\bigcup \gamma^K(K) = Sub$ as follows. Let $\approx, \preceq \subseteq \wp(K) \times \wp(K)$ be defined as $\mathcal{S}_1 \approx \mathcal{S}_2 \doteq \bigcup_{s_1 \in \mathcal{S}_1} \gamma^K(s_1) = \bigcup_{s_2 \in \mathcal{S}_2} \gamma^K(s_2)$ and $\mathcal{S}_1 \preceq \mathcal{S}_2 \doteq \bigcup_{s_1 \in \mathcal{S}_1} \gamma^K(s_1) \subseteq \bigcup_{s_2 \in \mathcal{S}_2} \gamma^K(s_2)$. Define $Z \doteq \wp(K)_{/\approx}$ and $\sqsubseteq^Z \doteq \preceq_{/\approx}$. It follows that $\langle Z, \sqsubseteq^Z \rangle$ is a complete lattice. Let $\gamma^Z : Z \nrightarrow \wp(Sub)$ be $\gamma^Z([\mathcal{S}]_{\approx}) \doteq \bigcup_{s \in \mathcal{S}} \gamma^K(s)$. It can be verified that γ^Z is an additive concretization function. Under this construction, $ex_{bw}^Z : VI^* \times Z \times Z \nrightarrow Z$ can be replaced with $ex_{bw}^Z : VI^* \times Z \times K \nrightarrow Z$. Thus, the design of the backward type analysis can be accomplished by (i) designing a pre-order $\langle \mathsf{VT}, \sqsubseteq^{\mathsf{VT}} \rangle$ of primitive type constraints and a monotone $\gamma^{\mathsf{VT}} : \mathsf{VT} \nrightarrow \wp(Sub)$ such that $\bigcup \gamma^{\mathsf{VT}}(\mathsf{VT}) = Sub$, (ii) constructing an abstract domain $\langle \mathsf{DVT}, \sqsubseteq^{\mathsf{DVT}} \rangle$ of type constraints and corresponding concretization function $\gamma^{\mathsf{DVT}} : \mathsf{DVT} \nrightarrow \wp(Sub)$, (iii) designing abstract operations uf_{bw}^{DVT}, ex_{bw}^{DVT}, $\exists_x^{\mathsf{DVT}}$, $\forall_x^{\mathsf{DVT}}$ and $\rho_{\boldsymbol{x} \nrightarrow \boldsymbol{y}}^{\mathsf{DVT}}$ and (iv) showing that they safely approximate (from below) corresponding operations on $\wp(Sub)$ with respect to γ^{DVT}.

4 Abstract Domain

The domain for the backward type analysis is the same as that for a forward type analysis [18]. A type system is first introduced on which the domain is based.

4.1 Types

A (monomorphic) type is a ground term constructed from a set $\mathsf{Cons} \cup \{\sqcap, \sqcup, \mathbf{1}, \mathbf{0}\}$ of type constructors. It is assumed that $(\mathsf{Cons} \cup \{\sqcap, \sqcup, \mathbf{1}, \mathbf{0}\}) \cap \Sigma = \emptyset$. The denotations of $\sqcap, \sqcup, \mathbf{1}$ and $\mathbf{0}$ are fixed whilst the denotations of those in Cons are determined by type rules [7]. The set of all types is then $\mathsf{RT} \doteq \mathsf{Term}(\mathsf{Cons} \cup \{\sqcap, \sqcup, \mathbf{1}, \mathbf{0}\}, \emptyset)$. A type parameter is a variable that ranges over RT. Let $\mathsf{Para}_m \doteq \{\beta_1, \cdots, \beta_m\}$ and $\mathsf{Para} \doteq \bigcup_{c \in \mathsf{Cons}} \mathsf{Para}_{arity(c)}$. A general type over Para_m is either a type parameter in Para_m or of the form $d(\beta_1, \cdots, \beta_k)$ with $d \in \mathsf{Cons}$ and $\beta_1, \cdots, \beta_k$ being different type parameters in Para_m. Note that general types are polymorphic. The set of all general types over Para_m is denoted GT_m and $\mathsf{GT} \doteq \bigcup_{c \in \mathsf{Cons}} \mathsf{GT}_{arity(c)}$. A type rule is of the form $c(\beta_1, \cdots, \beta_m) \rightarrow f(\tau_1, \cdots, \tau_n)$ where $f \in \Sigma$ and $\{c(\beta_1, \cdots, \beta_m), \tau_1, \cdots, \tau_n\} \subseteq \mathsf{GT}_m$. The set of all type rules is denoted Δ. It is assumed that each function symbol $f \in \Sigma$ occurs in at least one type rule in Δ and that each type constructor $c \in \mathsf{Cons}$ occurs in the lefthand side of at least one type rule in Δ.

Example 1. Let $\Sigma = \{0, s(), [\,], [\,|\,]\}$ and $\mathsf{Cons} = \{nat, even, odd, list()\}$. Then $\{\beta, list(\beta)\} \subseteq \mathsf{GT}$ and $list(even \sqcup list(odd)) \in \mathsf{RT}$. The following type rules define natural numbers, even numbers, odd numbers and lists. $\Delta = \{nat \rightarrowtriangle 0, nat \rightarrowtriangle s(nat), even \rightarrowtriangle 0, even \rightarrowtriangle s(odd), odd \rightarrowtriangle s(even), list(\beta) \rightarrowtriangle [\,],\ list(\beta) \rightarrowtriangle [\beta | list(\beta)]\}$. ∎

A type valuation is a mapping $\Bbbk = \{\beta_1 \mapsto R_1, \cdots, \beta_n \mapsto R_n\}$ where $\beta_i \in \mathsf{Para}$ and $R_i \in \mathsf{RT}$. The domain of $\Bbbk$ is defined $dom(\Bbbk) \doteq \{\beta_1, \cdots, \beta_n\}$. Let $\Bbbk(\beta) \doteq \mathbf{0}$ for any $\beta \notin dom(\Bbbk)$. Let VL denote the set of all type valuations. The application of a type valuation $\Bbbk$ to a general type τ is to replace simultaneously each occurrence of β in τ with $\Bbbk(\beta)$. A special type valuation $\top^{\mathsf{VL}}$ is introduced and defined $\top^{\mathsf{VL}}(\tau) \doteq \mathbf{1}$ for any general type τ. Let $ground(\Delta)$ be the set of all ground instances of rules in Δ plus rules of the form $\mathbf{1} \rightarrowtriangle f(\mathbf{1}, \cdots, \mathbf{1})$ for every $f \in \Sigma$. More exactly,

$$ground(\Delta) \doteq \{\Bbbk(\delta) \mid \delta \in \Delta \wedge \Bbbk \in (\mathsf{Para} \mapsto \mathsf{RT})\} \cup \{\mathbf{1} \mapsto f(\mathbf{1}, \cdots, \mathbf{1}) \mid f \in \Sigma\}$$

Given Δ, the set of terms denoted by a type is defined as follows.

$$\begin{array}{ll}
[\![\mathbf{1}]\!]_\Delta \doteq \mathsf{Term}(\Sigma, \mathcal{V}) & [\![\mathbf{0}]\!]_\Delta \doteq \emptyset \\
[\![R_1 \sqcap R_2]\!]_\Delta \doteq [\![R_1]\!]_\Delta \cap [\![R_2]\!]_\Delta & [\![R_1 \sqcup R_2]\!]_\Delta \doteq [\![R_1]\!]_\Delta \cup [\![R_2]\!]_\Delta \\
[\![c(R_1, \cdots, R_m)]\!]_\Delta \doteq & \\
\quad \{f(t_1, \cdots, t_n) \mid \exists (c(R_1, \cdots, R_m) \rightarrowtriangle f(T_1, \cdots, T_n)) \in ground(\Delta).\ t_i \in [\![T_i]\!]_\Delta\} &
\end{array}$$

$[\![\cdot]\!]_\Delta$ gives fixed denotations to $\sqcap, \sqcup, \mathbf{1}$ and $\mathbf{0}$. $\sqcap$ and $\sqcup$ are interpreted by $[\![\cdot]\!]_\Delta$ as set intersection and set union respectively. Every type in RT denotes a regular term language.

Example 2. Let Δ be that in Ex. 1. Then $[\![nat]\!]_\Delta = \{0, s(0), s(s(0)), \cdots\}$, $[\![list(\mathbf{0})]\!]_\Delta = \{[\,]\}$, and $[\![list(\mathbf{1})]\!]_\Delta = \{[\,], [x], \cdots\}$ where $x \in V$. ∎

Proposition 1. *Let $R \in \mathsf{RT}$. If $t \in [\![R]\!]_\Delta$ then $\xi(t) \in [\![R]\!]_\Delta$ for all $\xi \in Sub$. That is, types are closed under instantiation.*[1] ∎

Let R_1 and R_2 be two types. The set RT of types is ordered by $R_1 \sqsubseteq R_2$ iff $[\![R_1]\!]_\Delta \subseteq [\![R_2]\!]_\Delta$ and equivalence between types is defined by $R_1 \equiv R_2$ iff $[\![R_1]\!]_\Delta = [\![R_2]\!]_\Delta$.

4.2 Abstract Domain

The abstract domain is obtained by choosing a representation of type constraints informally introduced in section 2. Let $\mathsf{VT} \doteq (VI \mapsto \mathsf{RT})$ and $\sqcap^{\mathsf{VT}}$ and $\sqsubseteq^{\mathsf{VT}}$ be the point-wise extensions of the type constructor $\sqcap$ and the pre-order $\sqsubseteq$ respectively. Let $\top^{\mathsf{VT}} \doteq \{x \mapsto \mathbf{1} \mid x \in VI\}$. $\langle \mathsf{VT}, \sqsubseteq^{\mathsf{VT}} \rangle$ is a pre-order since $\langle \mathsf{RT}, \sqsubseteq \rangle$ is a pre-order. Members of VT represent primitive type constraints. For instance, if $VI = \{x, y\}$ then $\{x \mapsto nat, y \mapsto list(nat)\}$ constrains x to be of type nat and y to be of type $list(nat)$. The denotation of a primitive type constraint is given by $\gamma^{\mathsf{VT}} : \mathsf{VT} \mapsto \wp(Sub)$ defined as $\gamma^{\mathsf{VT}}(\mu) \doteq \{\theta \mid \forall x \in VI.(\theta(x) \in [\![\mu(x)]\!]_\Delta)\}$.

[1] This would change if set complement were a type constructor.

Observe that $\bigcup \gamma^{\mathsf{VT}}(\mathsf{VT}) = Sub$ since $\gamma^{\mathsf{VT}}(\top^{\mathsf{VT}}) = Sub$. A type constraint – a disjunction of primitive type constraints – is represented as a set containing exactly those primitive type constraints in the disjunction. For instance, if $VI = \{x, y\}$ and $\mathcal{S} = \{\{x \mapsto nat, y \mapsto list(nat)\}, \{x \mapsto list(nat), y \mapsto nat\}\}$ then $\mathcal{S}$ denotes $\{\theta \mid \theta(x) \in [\![nat]\!]_{\Delta} \wedge \theta(y) \in [\![list(nat)]\!]_{\Delta}\} \cup \{\xi \mid \xi(x) \in [\![list(nat)]\!]_{\Delta} \wedge \xi(y) \in [\![nat]\!]_{\Delta}\}$.

There may be many type constraints that denote the same set of substitutions. Type constraints that have the same denotation are identified as follows. Let relation $\preccurlyeq$ on $\wp(\mathsf{VT})$ be defined as $\mathcal{S}_1 \preccurlyeq \mathcal{S}_2 \doteq \bigcup_{\mu \in \mathcal{S}_1} \gamma^{\mathsf{VT}}(\mu) \subseteq \bigcup_{\nu \in \mathcal{S}_2} \gamma^{\mathsf{VT}}(\nu)$. Define relation $\approx$ on $\wp(\mathsf{VT})$ by $\mathcal{S}_1 \approx \mathcal{S}_2 \doteq (\mathcal{S}_1 \preccurlyeq \mathcal{S}_2) \wedge (\mathcal{S}_2 \preccurlyeq \mathcal{S}_1)$. Observe that $\approx$ is an equivalence relation on $\wp(\mathsf{VT})$.

Let $\mathsf{DVT} \doteq \wp(\mathsf{VT})_{/\approx}$ and $\sqsubseteq^{\mathsf{DVT}} \doteq \preccurlyeq_{/\approx}$. The abstract domain $\langle \mathsf{DVT}, \sqsubseteq^{\mathsf{DVT}}, \sqcup^{\mathsf{DVT}}, \sqcap^{\mathsf{DVT}}, \top^{\mathsf{DVT}}, \bot^{\mathsf{DVT}} \rangle$ is a complete lattice where $[\mathcal{S}_1]_{\approx} \sqcup^{\mathsf{DVT}} [\mathcal{S}_2]_{\approx} = [\mathcal{S}_1 \cup \mathcal{S}_2]_{\approx}$, $[\mathcal{S}_1]_{\approx} \sqcap^{\mathsf{DVT}} [\mathcal{S}_2]_{\approx} = [\{\{x \mapsto (\mu(x) \sqcap \nu(x)) \mid x \in VI\} \mid \mu \in \mathcal{S}_1 \wedge \nu \in \mathcal{S}_2\}]_{\approx}$, $\bot^{\mathsf{DVT}} = [\emptyset]_{\approx}$ and $\top^{\mathsf{DVT}} = [\{\top^{\mathsf{VT}}\}]_{\approx}$. Define $\gamma^{\mathsf{DVT}} : \mathsf{DVT} \mapsto \wp(Sub)$ by $\gamma^{\mathsf{DVT}}([\mathcal{S}]_{\approx}) \doteq \bigcup_{\mu \in \mathcal{S}} \gamma^{\mathsf{VT}}(\mu)$. $\gamma^{\mathsf{DVT}}(\mathsf{DVT})$ is a Moore family [18]. Thus, γ^{DVT} is a concretization function from $\langle \mathsf{DVT}, \sqsubseteq^{\mathsf{DVT}} \rangle$ to $\langle \wp(Sub), \subseteq \rangle$. Type constraints are closed under instantiation because types are closed under instantiation.

5 Abstract Operations for Backward Type Analysis

The design of the backward type analysis is now completed by defining those operations required by the abstract semantics outlined in section 3.2.

5.1 Simple Operations: $\exists_{\boldsymbol{x}}^{\mathsf{DVT}}$, $\forall_{\boldsymbol{x}}^{\mathsf{DVT}}$, $\rho_{\boldsymbol{x} \mapsto \boldsymbol{y}}^{\mathsf{DVT}}$ and $ex_{\boldsymbol{bw}}^{\mathsf{DVT}}$

The construction begins with the simple operations. Let $x \in VI$, $\boldsymbol{x} = \{x_1, \cdots, x_n\}$ a subset of VI and $\mu \in \mathsf{VT}$. The existential quantification operation $\exists_x^{\mathsf{VT}} : \mathsf{VT} \mapsto \mathsf{VT}$ returns $\{y \mapsto \mathbf{0} \mid y \in VI\}$ if $\mu(x)$ denotes the empty set of terms; otherwise, it removes the constraint on x from μ.

$$\exists_x^{\mathsf{VT}}(\mu) \doteq \begin{cases} \{y \mapsto \mathbf{0} \mid y \in VI\} & \text{if } \mu(x) \equiv \mathbf{0} \\ \mu \circ \{x \mapsto \mathbf{1}\} & \text{otherwise} \end{cases}$$

Let $\exists_{\boldsymbol{x}}^{\mathsf{VT}}(\mu) \doteq \exists_{x_1}^{\mathsf{VT}}(\exists_{x_2}^{\mathsf{VT}}(\cdots \exists_{x_n}^{\mathsf{VT}}(\mu)))$ and $\bar{\exists}_{\boldsymbol{x}}^{\mathsf{VT}}(\mu) \doteq \exists_{VI \setminus \boldsymbol{x}}^{\mathsf{VT}}(\mu)$. It follows that $\gamma^{\mathsf{VT}}(\exists_x^{\mathsf{VT}}(\mu) \sqcap^{\mathsf{VT}} \bar{\exists}_x^{\mathsf{VT}}(\mu)) = \gamma^{\mathsf{VT}}(\mu)$. The existential quantification operation $\exists_x^{\mathsf{DVT}} : \mathsf{DVT} \mapsto \mathsf{DVT}$ is defined as $\exists_x^{\mathsf{DVT}}([\mathcal{S}]_{\approx}) \doteq [\{\exists_x^{\mathsf{VT}}(\mu) \mid \mu \in \mathcal{S}\}]_{\approx}$. It removes the constraint on x from those primitive type constraints that describe a non-empty set of substitutions. Let $\exists_{\boldsymbol{x}}^{\mathsf{DVT}}(\phi) \doteq \exists_{x_1}^{\mathsf{DVT}}(\exists_{x_2}^{\mathsf{DVT}}(\cdots \exists_{x_n}^{\mathsf{DVT}}(\phi)))$ and $\bar{\exists}_{\boldsymbol{x}}^{\mathsf{DVT}}(\phi) \doteq \exists_{VI \setminus \boldsymbol{x}}^{\mathsf{DVT}}(\phi)$. The universal quantification operation $\forall_x^{\mathsf{DVT}} : \mathsf{DVT} \mapsto \mathsf{DVT}$ is defined as $\forall_x^{\mathsf{DVT}}([\mathcal{S}]_{\approx}) \doteq [\{\mu \mid (\mu \in \mathcal{S}) \wedge (\mu(x) \equiv \mathbf{1})\}]_{\approx}$. It removes primitive constraints that place any constraint on x. Let $\forall_{\boldsymbol{x}}^{\mathsf{DVT}}(\phi) \doteq \forall_{x_1}^{\mathsf{DVT}}(\forall_{x_2}^{\mathsf{DVT}}(\cdots \forall_{x_n}^{\mathsf{DVT}}(\phi)))$ and $\bar{\forall}_{\boldsymbol{x}}^{\mathsf{DVT}}(\phi) \doteq \forall_{VI \setminus \boldsymbol{x}}^{\mathsf{DVT}}(\phi)$. Existential and universal quantifications differ in their direction of approximation in that $\forall_x^{\mathsf{DVT}}(\phi) \sqsubseteq^{\mathsf{DVT}} \phi \sqsubseteq^{\mathsf{DVT}} \exists_x^{\mathsf{DVT}}(\phi)$. The renaming operation $\rho_{\boldsymbol{z} \mapsto \boldsymbol{y}}^{\mathsf{DVT}} : \mathsf{DVT} \mapsto \mathsf{DVT}$ is

defined as $\rho^{\mathsf{DVT}}_{\boldsymbol{z} \to \boldsymbol{y}}(\phi) \doteq [\boldsymbol{y}/\boldsymbol{z}]\phi$ where $[\boldsymbol{y}/\boldsymbol{z}]\phi$ results from substituting $\boldsymbol{y}$ for $\boldsymbol{z}$ simultaneously. The operation $ex^{\mathsf{DVT}}_{bw}(\boldsymbol{x}, \phi, \mu)$ is applied to $\phi \in \mathsf{DVT}$ and $\mu \in \mathsf{VT}$ when the execution of an atom $p(\boldsymbol{x})$ in a state satisfying ϕ can only succeed in a state satisfying $\bar{\exists}^{\mathsf{DVT}}_{\boldsymbol{x}}([\{\mu\}]_{\approx})$. The following definition of $ex^{\mathsf{DVT}}_{bw} : VI^* \times \mathsf{DVT} \times \mathsf{VT} \to \mathsf{DVT}$ ensures that $ex^{\mathsf{DVT}}_{bw}(\boldsymbol{x}, \phi, \mu)$ is a type constraint ψ such that the execution of $p(\boldsymbol{x})$ in any state satisfying ψ succeeds only in a state satisfying μ.

$$ex^{\mathsf{DVT}}_{bw}(\boldsymbol{x}, \phi, \mu) \doteq \phi \sqcap^{\mathsf{DVT}} \exists^{\mathsf{DVT}}_{\boldsymbol{x}}([\{\mu\}]_{\approx})$$

It is straightforward to show that the abstract quantification and renaming operations are correct, but the ex^{DVT}_{bw} operator is more subtle. Therefore let $\theta \in \gamma^{\mathsf{DVT}}(\phi \sqcap^{\mathsf{DVT}} \exists^{\mathsf{DVT}}_{\boldsymbol{x}}([\{\mu\}]_{\approx}))$ and suppose $\theta(p(\boldsymbol{x}))$ succeeds with a computed answer ζ. Since $\theta \in \gamma^{\mathsf{DVT}}(\phi)$, $\zeta \circ \theta \in \gamma^{\mathsf{DVT}}(\bar{\exists}^{\mathsf{DVT}}_{\boldsymbol{x}}([\{\mu\}]_{\approx}))$ by assumption. Since $\theta \in \gamma^{\mathsf{DVT}}(\exists^{\mathsf{DVT}}_{\boldsymbol{x}}([\{\mu\}]_{\approx}))$ and type constraints in DVT are closed under instantiation, $\zeta \circ \theta \in \gamma^{\mathsf{DVT}}(\exists^{\mathsf{DVT}}_{\boldsymbol{x}}([\{\mu\}]_{\approx}))$. Thus, $\zeta \circ \theta \in \gamma^{\mathsf{DVT}}([\{\mu\}]_{\approx})$ by the definitions of γ^{DVT} and γ^{VT}. The remaining operation $uf^{\mathsf{DVT}}_{bw} : Eqn \times \mathsf{DVT} \to \mathsf{DVT}$ is the most complex; it simulates the (reverse) effect of unification.

5.2 Backward Abstract Unification: uf^{DVT}_{bw}

In forward type analysis [18], abstract unification takes as inputs an equation E and a type constraint ϕ (an upper approximation) and produces as output a type constraint ψ (another upper approximation) which describes $mgu(\theta(E)) \circ \theta$ whenever ϕ describes θ. In backward type analysis, abstract unification takes as inputs an equation E and a type constraint ψ (a lower approximation) and produces as output a type constraint ϕ (another lower approximation) which describes θ whenever ψ describes $mgu(\theta(E)) \circ \theta$. Backward abstract unification is defined as a rewriting system which in turn is formulated in terms of the operations $type\_low$ (lower approximation to a type) and tc_low (lower approximation to a type constraint). These operations are themselves defined with the auxiliary operations introduced below.

Conjoining type valuations. Given two type valuations $\Bbbk_1$ and $\Bbbk_2$, $\curlywedge : \mathsf{VL} \times \mathsf{VL} \to \mathsf{VL}$ defined below gives another type valuation.

$$\Bbbk_1 \curlywedge \Bbbk_2 \doteq \begin{cases} \Bbbk_2 & \text{if } (\Bbbk_1 = \top^{\mathsf{VL}}); \\ \Bbbk_1 & \text{else if } (\Bbbk_2 = \top^{\mathsf{VL}}); \\ \{\beta \to \Bbbk_1(\beta) \sqcap \Bbbk_2(\beta) \mid \beta \in dom(\Bbbk_1) \cap dom(\Bbbk_2)\} & \text{otherwise} \end{cases}$$

By the definitions of $[\cdot]_{\Delta}$ and $\curlywedge$, it follows that $(\Bbbk_1 \curlywedge \Bbbk_2)(\tau) = \Bbbk_1(\tau) \sqcap \Bbbk_2(\tau)$ for any $\tau \in \mathsf{GT}$ and that $\curlywedge$ is commutative and associative with respect to $\equiv$.

Approximating type valuations. Given a type R, a general type τ, the function $tvs\_low : \mathsf{RT} \times \mathsf{GT} \to \wp(\mathsf{VL})$ defined below returns a set of type valuations that instantiate τ to types smaller than or equal to R.

$$tvs\_low(R,\tau) \doteq \begin{cases} \{\top^{\mathsf{VL}}\} & \text{if } R = \mathbf{1} \\ \{\{\tau \mapsto R\}\} & \text{else if } \tau \in \mathsf{Para} \\ tvs\_low(R_1,\tau) \cup tvs\_low(R_2,\tau) & \text{else if } R = R_1 \sqcup R_2 \\ \{\Bbbk_1 \curlywedge \Bbbk_2 \mid \Bbbk_i \in tvs\_low(R_i,\tau)\} & \text{else if } R = R_1 \sqcap R_2 \\ \{\{\beta_j \mapsto R_j \mid 1 \leq j \leq m\}\} & \text{else if } \begin{pmatrix} R = c(R_1,\cdots,R_m) \\ \tau = c(\beta_1,\cdots,\beta_m) \end{pmatrix} \\ \{\} & \text{otherwise} \end{cases}$$

The first two and the last two branches in the definition of tvs_low are obvious. The two recursive calls in the third branch returns two sets of type valuations members of which instantiate τ to types smaller than or equal to R_1 and R_2 respectively. Therefore, the union of two sets consists of type valuations that instantiate τ to types smaller than or equal to $R_1 \sqcup R_2$. The fourth branch applies $\curlywedge$ to each pair consisting of a type valuation from $tvs\_low(R_1,\tau)$ and a type valuation from $tvs\_low(R_2,\tau)$ resulting in a type valuation that instantiates R to a type smaller than or equal to $R_1 \sqcap R_2$.

Example 3. Let Δ be that in Ex. 1. Then $tvs\_low(nat,\beta) = \{\{\beta \mapsto nat\}\}$ and $tvs\_low(list(nat) \sqcup list(list(nat)), list(\beta)) = \{\{\beta \mapsto nat\}\} \cup \{\{\beta \mapsto list(nat)\}\} = \{\{\beta \mapsto nat\}, \{\beta \mapsto list(nat)\}\}$. ∎

Lemma 1. *Let R be a type and τ a general type. Then* $[\sqcup_{\Bbbk \in tvs\_low(R,\tau)} \Bbbk(\tau)]_\Delta \subseteq [R]_\Delta$. ∎

Approximating types. Given a primitive type constraint μ and a term t, $type\_low : \mathsf{VT} \times \mathsf{Term}(\Sigma, VI) \mapsto \mathsf{RT}$ gives a type that describes a subset of those terms $\theta(t)$ for which μ describes θ.

$$type\_low(\mu,t) \doteq \begin{cases} \mu(t) & \text{if } t \in VI \\ \sqcup \left\{ (\Bbbk_1 \curlywedge \cdots \curlywedge \Bbbk_n)(\tau) \;\middle|\; \begin{array}{c} R_i = type\_low(\mu,t_i) \\ \wedge\ (\tau \rightarrow f(\tau_1,\cdots,\tau_n)) \in \Delta \\ \wedge\ \Bbbk_i \in tvs\_low(R_i,\tau_i) \end{array} \right\} & \text{if } t = f(t_1,\cdots,t_n) \end{cases}$$

Lemma 2. *Let $\mu \in \mathsf{VT}$, $t \in \mathsf{Term}(\Sigma, VI)$ and $\theta \in Sub$. If $\theta(t) \in [type\_low(\mu,t)]_\Delta$ then $\theta \in \gamma^{\mathsf{VT}}(\mu)$.* ∎

Example 4. Continuing with Ex.3, let $\mu = \{t_1 \mapsto nat, t_2 \mapsto list(nat) \sqcup list(list(nat))\}$ and $t = [t_1|t_2]$. Let $R_1 = type\_low(\mu,t_1)$ and $R_2 = type\_low(\mu,t_2)$. Then $R_1 = nat$ and $R_2 = list\ (nat) \sqcup list\ (list(nat))$. The only applicable type rule is $list(\beta) \rightarrow [\beta|\ list(\beta)]$. By Ex. 3, $tvs\_low(R_1,\beta) = \{\{\beta \mapsto nat\}\}$ and $tvs\_low(R_2, list(\beta)) = \{\{\beta \mapsto nat\}, \{\beta \mapsto list(nat)\}\}$. Let $\Bbbk_1 = \{\beta \mapsto nat\}$, $\Bbbk_{21} = \{\beta \mapsto nat\}$ and $\Bbbk_{22} = \{\beta \mapsto list\ (nat)\}$. Then $tvs\_low(R_1,\beta) = \{\Bbbk_1\}$ and $tvs\_low(\ R_2, list(\beta)) = \{\Bbbk_{21}, \Bbbk_{22}\}$. Thus, $type\_low(\mu,[t_1|t_2]) = \sqcup\{(\Bbbk_1 \curlywedge \Bbbk_{21})(list(\beta)), (\Bbbk_1 \curlywedge \Bbbk_{22})(list(\beta))\} = list(nat) \sqcup list(nat \sqcap list(nat)) \equiv list(nat)$ since $nat \sqcap list(nat) \equiv \mathbf{0}$. Observe that if $\theta([t_1|t_2]) \in [type\_low(\mu,[t_1|t_2])]_\Delta = [list(nat)]_\Delta$ then $\theta \in \gamma^{\mathsf{VT}}(\mu)$. ∎

Approximating type constraints. Let $R \in \mathsf{RT}$ and $t \in \mathsf{Term}(\Sigma, VI)$. The operation $tc\_low : \mathsf{RT} \times \mathsf{Term}(\Sigma, VI) \mapsto \mathsf{DVT}$ defined below gives a type constraint ϕ such that $\theta(t) \in [\![R]\!]_\Delta$ for every θ described by ϕ:

$$
\begin{aligned}
tc\_low(R, x) &\doteq [\{\top^{\mathsf{VT}} \circ \{x \mapsto R\}\}]_{\approx} \\
tc\_low(R_1 \sqcup R_2, t) &\doteq tc\_low(R_1, t) \ \sqcup^{\mathsf{DVT}} \ tc\_low(R_2, t) \\
tc\_low(R_1 \sqcap R_2, t) &\doteq tc\_low(R_1, t) \ \sqcap^{\mathsf{DVT}} \ tc\_low(R_2, t) \\
tc\_low(R, f(t_1, \cdots, t_n)) &\doteq \sqcup^{\mathsf{DVT}}_{(R \rightarrow f(R_1, \cdots, R_n)) \in ground(\Delta)} \ \sqcap^{\mathsf{DVT}}_{1 \le i \le n} \ tc\_low(R_i, t_i)
\end{aligned}
$$

Observe that if $R = \mathbf{0}$ and $t = f(t_1, \cdots, t_n)$ then $tc\_low(R, t) = \bot^{\mathsf{DVT}}$ via the fourth definition because $\mathbf{0} \rightarrow f(R_1, \cdots, R_n)$ is not in $ground(\Delta)$. Also note that if $R = \mathbf{1}$ and $t = f(t_1, \cdots, t_n)$ then $tc\_low(R, t) = \top^{\mathsf{DVT}}$ since $\mathbf{1} \rightarrow f(\mathbf{1}, \cdots, \mathbf{1})$ is in $ground(\Delta)$.

Example 5. Let Δ be that in Ex. 1 and $VI = \{x, y\}$. Then $tc\_low(list(odd), [x|y]) = tc\_low(odd, x) \ \sqcap^{\mathsf{DVT}} \ tc\_low(list(odd), y) = [\{\{x \mapsto odd, y \mapsto \mathbf{1}\}\}]_{\approx} \ \sqcap^{\mathsf{DVT}} [\{\{x \mapsto \mathbf{1}, \ y \mapsto list(odd)\}\}]_{\approx} = [\{\{x \mapsto odd, y \mapsto list(odd)\}\}]_{\approx}$ and $tc\_low(list(even), [x|y]) = [\{\{x \mapsto even, y \mapsto list(even)\}\}]_{\approx}$. Thus, $tc\_low(list(odd) \sqcup list(even), [x|y]) = tc\_low(list(odd), [x|y]) \ \sqcup^{\mathsf{DVT}} tc\_low(list(even), [x|y]) = [\{\{x \mapsto odd, y \mapsto list(odd)\}, \{x \mapsto even, y \mapsto list(even)\}\}]_{\approx}$. ∎

Lemma 3. *Let $R \in \mathsf{RT}$ and $t \in \mathsf{Term}(\Sigma, VI)$. Then $\theta(t) \in R$ for any $\theta \in \gamma^{\mathsf{DVT}}(tc\_low(R, t))$.* ∎

Propagating type constraints backwards by rewriting. The abstract unification is based on a rewriting relation that propagates a type constraint ψ over an equational constraint E in solved form. The relation $\rightsquigarrow \subseteq (\mathsf{VT} \times Eqn) \times (\mathsf{VT} \times Eqn)$ is constructed from from $type\_low$ and tc_low and is defined as follows:

$$\langle \mu, E \rangle \rightsquigarrow \langle \mu, \emptyset \rangle \tag{1}$$

$$\langle \mu, E \cup \{x = t\} \rangle \rightsquigarrow \langle (\top^{\mathsf{VT}} \circ \{x \mapsto type\_low(\mu, t)\}) \sqcap^{\mathsf{VT}} \exists^{\mathsf{VT}}_{Var(t)}(\mu), E \rangle \tag{2}$$

$$\langle \mu, E \cup \{x = t\} \rangle \rightsquigarrow \langle \exists^{\mathsf{VT}}_{x}(\mu) \sqcap^{\mathsf{VT}} \nu, E \rangle \text{ where } \nu \in \mathcal{S} \text{ and } [\mathcal{S}]_{\approx} = tc\_low(\mu(x), t) \tag{3}$$

The relation $\rightsquigarrow^*$ is the reflexive and transitive closure of $\rightsquigarrow$. The first rule is justified because $\gamma^{\mathsf{VT}}(\mu)$ is downward closed and hence $\xi = mgu(\theta(E)) \circ \theta$ is described by μ if θ is described by μ. Each rewriting step by the other rules infers a primitive type constraint μ' from μ and an equation $x = t$ such that if μ' describes θ then μ describes $\xi = mgu(\theta(x), \theta(t)) \circ \theta$. Consider rule (2) first. Let $R = type\_low(\mu, t)$ and $\theta(t) \in [\![R]\!]_\Delta$. Then $\theta(y) \in \gamma^{\mathsf{VT}}(\mu(y))$ for all $y \in Var(t)$ by Lemma 2, in other words, if the x is constrained to be of type R then the type constraints on $y \in Var(t)$ can be removed from μ since they are implied by the type of x. Now consider rule (3). Let $[\mathcal{S}]_{\approx} = tc\_low(\mu(x), t)$, $\nu \in \mathcal{S}$ and $\theta \in \gamma^{\mathsf{VT}}(\nu)$. Then $\theta(t) \in \gamma^{\mathsf{VT}}(\mu(x))$ by Lemma 3. Thus, the type constraint on x in μ is implied by ν, hence can be removed if ν is asserted.

Example 6. Let $VI = \{x, y, h, l_1, l_2\}$, $E = \{x = [h|l_1], y = [h|l_2]\}$ and $\mu_0 = \{x \mapsto list(nat),\ y \mapsto list(nat), h \mapsto nat, l_1 \mapsto list(nat), l_2 \mapsto list(nat)\}$. Then $\langle \mu_0, E\rangle \rightsquigarrow^{(x=[h|l_1],2)} \langle \mu_1, \{y = [h|l_2]\}\rangle \rightsquigarrow^{(y=[h|l_2],2)} \langle \mu_2, \emptyset\rangle$ and $\langle \mu_0, E\rangle \rightsquigarrow^{(y=[h|l_2],3)} \langle \mu_3, \{x = [h|l_1]\}\rangle \rightsquigarrow^{(x=[h|l_1],2)} \langle \mu_4, \emptyset\rangle$ where each step is labelled with the selected equation and the selected rewriting rule and

$$\mu_1 = \{x \mapsto list(nat), y \mapsto list(nat), h \mapsto \mathbf{1}, l_1 \mapsto \mathbf{1}, l_2 \mapsto list(nat)\}$$
$$\mu_2 = \{x \mapsto list(nat), y \mapsto list(nat), h \mapsto \mathbf{1}, l_1 \mapsto \mathbf{1}, l_2 \mapsto \mathbf{1}\}$$
$$\mu_3 = \{x \mapsto list(nat), y \mapsto \mathbf{1}, h \mapsto nat, l_1 \mapsto list(nat), l_2 \mapsto list(nat)\}$$
$$\mu_4 = \{x \mapsto list(nat), y \mapsto \mathbf{1}, h \mapsto \mathbf{1}, l_1 \mapsto \mathbf{1}, l_2 \mapsto list(nat)\}$$

Moreover $\langle \mu_1, \{y = [h|l_2]\}\rangle \not\rightsquigarrow \langle \mu_4, \emptyset\rangle$ and $\langle \mu_3, \{x = [h|l_1]\}\rangle \not\rightsquigarrow \langle \mu_2, \emptyset\rangle$. Observe that $\gamma^{\mathsf{VT}}(\mu_2) \not\subseteq \gamma^{\mathsf{VT}}(\mu_4)$ and $\gamma^{\mathsf{VT}}(\mu_4) \not\subseteq \gamma^{\mathsf{VT}}(\mu_2)$. This shows that $\rightsquigarrow$ does not have the diamond property with respect to the selection of the equation. ∎

Proposition 2. *Let $E \in Eqn$ and $\mu, \nu \in \mathsf{VT}$ such that $\langle \mu, E\rangle \rightsquigarrow^* \langle \nu, \emptyset\rangle$. Then $mgu(\theta(E)) \circ \theta \in \gamma^{\mathsf{VT}}(\mu)$ for every $\theta \in \gamma^{\mathsf{VT}}(\nu)$.* ∎

Backward Abstract Unification. The backward abstract unification operation $uf_{bw}^{\mathsf{DVT}} : Eqn \times \mathsf{DVT} \mapsto \mathsf{DVT}$ is defined

$$uf_{bw}^{\mathsf{DVT}}(E, [\mathcal{S}]_{\approx}) \doteq [\{\nu \mid \exists \mu \in \mathcal{S}.(\langle \mu, E\rangle \rightsquigarrow^* \langle \nu, \emptyset\rangle)\}]_{\approx}$$

$uf_{bw}^{\mathsf{DVT}}(E, [\mathcal{S}]_{\approx})$ rewrites $\langle \mu, E\rangle$ in every possible way for primitive constraint μ in $\mathcal{S}$ and collects the resulting primitive constraints. Proposition 2 can be interpreted as saying that correctness is not compromised by barring either rule (1), rule (2) or rule (3). Maximising the number of rules used maximises $uf_{bw}^{\mathsf{DVT}}(E, [\mathcal{S}]_{\approx})$ and therefore maximises precision by minimising the resulting type constraint.

Example 7. Continuing with Ex. 6, let

$$\mu_5 = \{x \mapsto \mathbf{1}, y \mapsto list(nat), h \mapsto nat, l_1 \mapsto list(nat), l_2 \mapsto list(nat)\}$$
$$\mu_6 = \{x \mapsto list(nat), y \mapsto list(nat), h \mapsto \mathbf{1}, l_1 \mapsto list(nat), l_2 \mapsto \mathbf{1}\}$$
$$\mu_7 = \{x \mapsto \mathbf{1}, y \mapsto list(nat), h \mapsto \mathbf{1}, l_1 \mapsto list(nat), l_2 \mapsto \mathbf{1}\}$$
$$\mu_8 = \{x \mapsto \mathbf{1}, y \mapsto \mathbf{1}, h \mapsto nat, l_1 \mapsto list(nat), l_2 \mapsto list(nat)\}$$

Then $\langle \mu_0, E\rangle \rightsquigarrow^* \langle \mu_i, \emptyset\rangle$ for each $0 \leq i \leq 8$. It can be verified that there is no other primitive type constraint ν such that $\langle \mu_0, E\rangle \rightsquigarrow^* \langle \nu, \emptyset\rangle$. Thus, by the definition of uf_{bw}^{DVT}, $uf_{bw}^{\mathsf{DVT}}(E, [\{\mu_0\}]_{\approx}) = [\{\mu_i \mid 0 \leq i \leq 8\}]_{\approx} = [\{\mu_2, \mu_4, \mu_7, \mu_8\}]_{\approx}$ Since $\mu_0 \sqsubseteq^{\mathsf{VT}} \mu_1 \sqsubseteq^{\mathsf{VT}} \mu_2$, $\mu_3 \sqsubseteq^{\mathsf{VT}} \mu_4$, $\mu_5 \sqsubseteq^{\mathsf{VT}} \mu_7$ and $\mu_6 \sqsubseteq^{\mathsf{VT}} \mu_7$. ∎

The following result states the correctness of uf_{bw}^{DVT}, which together with correctness of $ex_{bw}^{\mathsf{DVT}}, \exists_x^{\mathsf{DVT}}, \forall_x^{\mathsf{DVT}}$ and $\rho_{\boldsymbol{x} \mapsto \boldsymbol{y}}^{\mathsf{DVT}}$ implies the correctness of the backward type analysis.

Theorem 1. *Let $E \in Eqn$ and $\phi \in \mathsf{DVT}$. Then $mgu(\theta(E)) \circ \theta \in \gamma^{\mathsf{DVT}}(\phi)$ for every $\theta \in \gamma^{\mathsf{DVT}}(uf_{bw}^{\mathsf{DVT}}(E, \phi))$.* ∎

6 Prototype Analyser and Examples

In order to evaluate the usefulness of the analysis presented in sections 4 and 5, a backward type analyser has been constructed for inferring type pre-conditions. The analyser is coded in SICStus Prolog 3.8.3. The analyser takes, as input, a program written in a declarative subset of ISO Prolog, type definitions for function symbols occurring in the program, type assertions for selected user-defined predicates, and a set of initial demands each consisting of an atom paired with a type post-condition. The analyser outputs a type constraint that is a type pre-condition for each initial demand. The safety result of the analysis ensures that if the atom in an initial demand is executed in a state satisfying the inferred pre-condition for the demand, then the execution will not violate any type assertion and the post-condition in the demand will be satisfied by any state in which the execution succeeds. The analyser supports pre-defined types such as $atom$, $float$, int, num and $string$ with usual meanings.

The implementation of abstract operations follows section 5 closely. The top-level of the analyser was straightforward to implement as it is essentially a fixpoint computation. The only subtlety is in handling the builtins. For each builtin, it is necessary to specify a type assertion that is strong enough for avoiding a type error. This is a lower approximation (the *Assertion* column of table 1). It is also necessary to specify an operation that transforms a type post-condition for a builtin to a type pre-condition. These operations output lower approximations (the *Operation* column of table 1). Table 1 displays type constraints instead of their representations in DVT for improved readability.

Type assertions for some builtins are given in Table 1; most are easily verified but some warrant further explanation. Unification `=/2` is modelled by the abstract unification operation uf_{bw}^{DVT}. Builtins such as `>/2` and $put/1$ that never instantiate their arguments, are modelled by the identity function $\lambda\phi.\phi$ since any type constraint that holds when the builtin succeeds must also hold before the builtin is called. The builtin $fail/0$ is modelled by the constant function that always returns $true$ ($\top^{\mathsf{DVT}}$) since any type post-condition of $fail/0$ is vacuously satisfied.

Consider a builtin to which a call $p(\boldsymbol{x})$ will definitely instantiate x in $\boldsymbol{x}$ to a term of type R upon success. Let μ be a conjunct in the type post-condition for the call. Observe that μ is a type pre-condition for the demand $\langle p(\boldsymbol{x}), \mu\rangle$. This type pre-condition, however, can be weakened if $R \sqsubseteq \mu(x)$. Specifically, if $\exists_x^{\mathsf{VT}}.\mu$ holds before the execution of the call and $R \sqsubseteq \mu(x)$ then μ holds upon success of the execution. In other words, $\exists_x^{\mathsf{VT}}.\mu$ is a type pre-condition for the demand $\langle p(\boldsymbol{x}), \mu\rangle$ if $R \sqsubseteq \mu(x)$. Thus define $relax : \mathsf{VT} \times VI \times \mathsf{RT}$ by

$$relax(\mu, x, R) \doteq \text{ if } R \sqsubseteq \mu(x) \text{ then } \exists_x^{\mathsf{VT}}.\mu \text{ else } \mu$$

and $relax : \mathsf{DVT} \times VI \times \mathsf{RT}$ by $relax([\mathcal{S}]_{\approx}, x, R) \doteq [\{relax(\mu, x, R) \mid \mu \in \mathcal{S}\}]_{\approx}$. The operation for the builtin $p(\boldsymbol{x})$ transforms ϕ to $relax(\phi, x, R)$ for each x in $\boldsymbol{x}$ that is definitely of type R upon success of the execution of $p(\boldsymbol{x})$. For instance, the operation for $is(x_1, x_2)$ is $\lambda\phi.relax(\phi, x_1, num)$ since x_1 is definitely of type num upon success of $is(x_1, x_2)$. Note that the execution of $is(x_1, x_2)$ does not instantiate x_2. The operation for $read(x_1)$ is $\lambda\phi.\phi$ since x_1 does not necessarily belong to any type other than $\mathbf{1}$ upon the success of $read(x_1)$.

Table 1. Type assertions and abstract operations for builtins where $\phi_1 = (x_1 \in atom \sqcup string \sqcup list(int))$, $\phi_2 = (x_2 \in num)$, $\phi_3 = (x_1 \in int)$, $\phi_4 = (x_1 \in num \wedge x_2 \in num)$, $\phi_5 = (x_1 \in atom \sqcup int \vee x_2 \in string)$, $f_1 = \lambda\phi.relax(relax(\phi, x_1, list(\mathbf{1})), x_2, int)$ and $f_2 = \lambda\phi.relax(relax(\phi, x_1, atom \sqcup int), x_2, string)$.

Builtin	*Assertion*	*Operation*
$abort, fail, false$	$true$	$\lambda\phi.true$
!, x_1@<x_2, x_1@>x_2, x_1=<@x_2, x_1@>=x_2, x_1==x_2, x_1\==x_2, x_1\=x_2, $compound(x_1)$, $display(x_1)$, $ground(x_1)$, $listing$, $listing(x_1)$, nl, $nonvar(x_1)$, $portray\_clause(x_1)$, $print(x_1)$, $read(x_1)$, $repeat$, $true$, $var(x_1)$, $write(x_1)$, $writeq(x_1)$, $float(x_1)$, $string(x_1)$, $atom(x_1)$, $atomic(x_1)$, $integer(x_1)$, $number(x_1)$	$true$	$\lambda\phi.true$
x_1=x_2	$true$	$\lambda\phi.uf_{bw}^{\mathsf{DVT}}(\{x_1 = x_2\}, \phi)$
$format(x_1)$, $format(x_1, x_2)$, $format(x_0, x_1, x_2)$	ϕ_1	$\lambda\phi.\phi$
$is(x_1, x_2)$	ϕ_2	$\lambda\phi.relax(\phi, x_1, num)$
$erase(x_1)$, $put(x_1)$, $tab(x_1)$	ϕ_3	$\lambda\phi.\phi$
x_1<x_2, $x_1 > x_2$, x_1=<x_2, x_1>=x_2, x_1=:=x_2, x_1=\=x_2	ϕ_4	$\lambda\phi.\phi$
$length(x_1, x_2)$	$true$	f_1
$compare(x_1, x_2, x_3)$	$true$	$\lambda\phi.relax(\phi, x_1, atom)$
$name(x_1, x_2)$	ϕ_5	f_2

The analyser forces termination by limiting the number of types that may occur in demands and thereby limiting the number of demands. This is achieved by depth-k abstraction – a technique that is frequently applied in type analysis [1,3,18]. Sub-terms at depth k in a type R are replaced with $\mathbf{0}$ (rather than $\mathbf{1}$), resulting a type R' such that $R' \sqsubseteq R$. Thus, application of depth-k abstraction to types in the post-condition ϕ in a demand $\langle p(\boldsymbol{x}), \phi\rangle$ obtains a post-condition ϕ' which is at least as strong as ϕ. Safety is preserved because a pre-condition for the stronger $\langle p(\boldsymbol{x}), \phi'\rangle$ is also a pre-condition for the weaker $\langle p(\boldsymbol{x}), \phi\rangle$.

The analyser has been applied to some standard Prolog benchmarks which can be found at http://www.oakland.edu/∼l2lu/Benchmarks-BT.zip. Inferred type signatures for the predicates in the smaller benchmarks, are given in table 2 which also displays type constraints instead of their representations in DVT. The type assertions for each benchmark program are exactly those for the builtins that are called in the program although type assertions may be provided for user-defined predicates as well. The type signatures were obtained by analysing the program with a set of initial demands – one demand with the type post-condition $true$ for each predicate in the program. The results have been verified by hand and, though sometimes surprising, appear to be optimal. For instance, the first conjunct in the inferred type signature ϕ_2 for predicate $partition/4$ indicates that the execution of $partition(x_1, x_2, x_3, x_4)$ is free from type errors if x_2 is a number, and x_3 and x_4 are lists of numbers when the execution starts. The analyser can thus infer type signatures that may well be missed by manual inspection. The main weakness of the prototype is that its front end does not currently support control features such as ; and -> or the meta-programming builtins, though this is not a fundamental limitation of the analysis itself.

Table 2. Precision of the Backward Type Analysis where $\phi_1 = (x_1 \in num \wedge x_4 \in tree(num)) \vee (x_1 \in num \wedge x_2 \in tree(num) \wedge x_3 \in tree(num))$, $\phi_2 = (x_2 \in num \wedge x_3 \in list(num) \wedge x_4 \in list(num)) \vee (x_1 \in list(num) \wedge x_2 \in num)$ and $\phi_3 = (x_1 \in num \wedge x_3 \in tree(num)) \vee (x_1 \in num \wedge x_2 \in tree(num))$.

Benchmark	*Predicate*	*Inferred Type Signature*
merge	$merge(x_1, x_2, x_3)$	$(x_1 \in list(num) \wedge x_2 \in list(num))$
heapify	$adjust(x_1, x_2, x_3, x_4)$	ϕ_1
	$heapify(x_1, x_2)$	$(x_1 \in tree(num))$
	$greater(x_1, x_2)$	$(x_1 \in num \wedge x_2 \in tree(num))$
quicksort using stack	$partition(x_1, x_2, x_3, x_4)$	ϕ_2
	$iqsort\_aux(x_1, x_2, x_3)$	$(x_1 \in list(num))$
	$iqsort(x_1, x_2)$	$(x_1 \in list(num))$
quicksort	$append(x_1, x_2, x_3)$	$true$
	$partition(x_1, x_2, x_3, x_4)$	ϕ_2
	$quicksort(x_1, x_2)$	$(x_1 \in list(num))$
quicksort using difference list	$partition(x_1, x_2, x_3, x_4)$	ϕ_2
	$quicksort\_dl(x_1, x_2)$	$(x_1 \in list(num))$
	$quicksort(x_1, x_2)$	$(x_1 \in list(num))$
treesort	$tree\_to\_list\_aux(x_1, x_2, x_3)$	$true$
	$tree\_to\_list(x_1, x_2)$	$true$
	$insert(x_1, x_2, x_3)$	ϕ_3
	$insert\_list(x_1, x_2, x_3)$	$(x_1 \in list(num) \wedge x_2 \in tree(num))$
	$list\_to\_tree(x_1, x_2)$	$(x_1 \in list(num))$
	$treesort(x_1, x_2)$	$(x_1 \in list(num))$
lookup	$lookup(x_1, x_2, x_3)$	$x_1 \in num \wedge x_2 \in dictionary(num, \mathbf{1})$
exp	$exp(x_1, x_2, x_3)$	$(x_1 \in num \wedge x_2 \in num)$
factorial	$factorial(x_1, x_2)$	$x_1 \in num$

7 Related Work

The literature on types in logic programming is vast so this section provides some initial pointers to related work on type analysis. Analysis can be performed either with [1,2,3,11, 13,14,17,18] or without [4,8,9,10,12,20] type definitions provided by the programmer. The former are easy for the programmer to understand whereas the latter are useful in compiler optimisation but can be more difficult for the programmer to interpret.

All the above type analyses propagate type information in the direction of program execution and compute upper approximations to the set of reachable program states. In contrast, the backward type analysis presented in this paper propagates type information in the reverse direction of program execution and computes lower approximations to the set of program states from which the execution will not violate any type assertions.

In a related work [15], the authors present an abstract semantics for backward analysis of logic programs and specialise it to the groundness domain Pos [5,19] to infer safe modes for queries which ensure that the program will not generate an instantiation error. Analysis is performed by first computing an upper approximation to the success set of the program and then a lower approximation to the set of programs states (substitutions) that will not violate any moding requirement. Both phases of analysis require the domain

to be equipped with a (computable) intuitionistic implication operator. Although [3] presents a type domain that is a complete Heyting algebra, its intuitionistic implication operator is left unspecified.

Pedreschi and Ruggieri [21] develop a calculus of weakest pre-conditions and weakest liberal pre-conditions, the latter of which is essentially a reformulation of Hoare's logic. Weakest liberal pre-conditions are characterised as the greatest fixpoint of a co-continuous operator on the space of interpretations. The work is motivated by, among other things, the desire to infer the absence of ill-typed arithmetic. Our work takes these ideas forward to show how abstract interpretation can infer weakest liberal pre-conditions.

8 Summary

A novel backward type analysis for logic programs has been presented. The analysis generalises type checking and is able to infer type constraints for predicates in a program that, if satisfied, guarantee that the execution of the program will not violate any type assertions. The analysis can relieve the programmer from the tedium of declaring all types, infer valuable program documentation and also aid in debugging. Algorithms for the domain operations have been specified that can be translated directly into implementation and a prototype implementation has demonstrated that useful type information can be inferred. Contrary to what was first thought [15], the work also shows that a domain for backward type analysis need not be a complete Heyting algebra.

Acknowledgement. The work of King was supported, in part, by EPSRC grant GR/MO8769; the work of Lunjin Lu is supported, in part, by the National Science Foundation (CCR-0131862).

References

1. R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Science of Computer Programming*, 19(3):133–181, 1992.
2. M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of Prop. In *Proceedings of International Static Analysis Symposium*, pages 281–297. Springer-Verlag, 1994.
3. M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. *Theoretical Computer Science*, 238:131–159, 2000.
4. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 22(3):179–208, 1995.
5. A. Cortesi, G. Filé, and W. Winsborough. Optimal groundness analysis using propositional logic. *Journal of Logic Programming*, 27(2):137–168, 1996.
6. P. Cousot and R. Cousot. Abstract interpretation: a unified framework for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of Principles of Programming Languages*, pages 238–252. The ACM Press, 1977.
7. P.W. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–189. The MIT Press, 1992.

8. J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In *Proceedings of International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.
9. N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Proceedings of Principles of Programming Languages*, pages 197–209. The ACM Press, 1990.
10. N. Heintze and J. Jaffar. Semantic types for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 141–155. The MIT Press, 1992.
11. K. Horiuchi and T. Kanamori. Polymorphic type inference in Prolog by abstract interpretation. In *Proceedings of Conference on Logic Programming*, pages 195–214. Springer, 1988.
12. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(1–4):205–258, 1992.
13. T. Kanamori and K. Horiuchi. Type inference in Prolog and its application. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 704–707. Morgan Kaufmann, 1985.
14. T. Kanamori and T. Kawamura. Abstract interpretation based on OLDT resolution. *Journal of Logic Programming*, 15(1 & 2):1–30, 1993.
15. A. King and L. Lu. A backward analysis for constraint logic programs. *Theory and Practice of Logic Programming*, 2(4&5):517–547, 2002.
16. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
17. L. Lu. A polymorphic type analysis in logic programs by abstract interpretation. *Journal of Logic Programming*, 36(1):1–54, 1998.
18. L. Lu. A precise type analysis of logic programs. In *Proceedings of International Conference on Principles and Practice of Declarative Programming*, pages 214–225. The ACM Press, 2000.
19. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.
20. P. Mishra. Towards a theory of types in Prolog. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 289–298. The IEEE Computer Society Press, 1984.
21. D. Pedreschi and S. Ruggieri. Weakest preconditions for pure Prolog programs. *Information Processing Letters*, 67(3):145–150, 1998.

More Precise Yet Efficient Type Inference for Logic Programs

Claudio Vaucheret and Francisco Bueno

Technical University of Madrid (UPM), Spain
{claudio,bueno}@fi.upm.es

Abstract. Type analyses of logic programs which aim at inferring the types of the program being analyzed are presented in a unified abstract interpretation-based framework. This covers most classical abstract interpretation-based type analyzers for logic programs, built on either top-down or bottom-up interpretation of the program. In this setting, we discuss the widening operator, arguably a crucial one. We present a new widening which is more precise than those previously proposed. Practical results with our analysis domain are also presented, showing that it also allows for efficient analysis.

1 Introduction

In type analyses, the widening operation has much influence in the results. If the widening is too aggressive in making approximations then the analysis results may be too imprecise. On the other hand, if it is not sufficiently aggressive then the analysis may become too inefficient.

Widening operators are aimed at identifying the recursive structure of the types being inferred. All widening operators already proposed in the literature are based on locating type nodes with the same functors, which are possible sources of recursion. However, they disregard whether such nodes come in fact from a recursive structure in the program or not. This may originate an unnecessary loss of precision, since the widening result may then impose a recursive structure on the resulting type in argument positions where the concrete program is in fact not recursive. We propose a widening operator to try to remedy this problem.

We present our widening operator for regular type inference in an analysis framework based on abstract interpretation of the program. In order for the paper to be self contained, we first revisit regular types (Section 2) and, in particular, deterministic ones. We focus on deterministic types for ease of presentation; however, there is nothing in our widening which prevents it to be applicable also to non-deterministic types. The abstract interpretation framework is set up in Section 3. Section 4 reviews previous widenings in the literature, and Section 5 presents ours. In Section 6 experimental results are presented, and Section 7 concludes and discusses future work.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 102–116, 2002.
© Springer-Verlag Berlin Heidelberg 2002

2 Regular Types

A *regular type* [3] is a type representing a class of terms that can be described by a regular term grammar. A *regular term grammar*, or grammar for short, describes a set of finite terms constructed from a finite alphabet $\mathcal{F}$ of *ranked function symbols* or *functors*. A grammar $G = (S, \mathcal{T}, \mathcal{F}, \mathcal{R})$ consists of a set of non-terminal symbols $\mathcal{T}$, one distinguished symbol $S \in \mathcal{T}$, and a finite set $\mathcal{R}$ of productions $T \longrightarrow rhs$, where $T \in \mathcal{T}$ is a non-terminal and the right hand side rhs is either a non-terminal or a term $f(T_1, \ldots, T_n)$ constructed from an n-ary function symbol $f \in \mathcal{F}$ and n non-terminals.

The non-terminals $\mathcal{T}$ are *types* describing (ground) terms built from the functors in $\mathcal{F}$. The concretization $\gamma(T)$ of a non-terminal T is the set of terms derivable from its productions, that is,

$$\gamma(T) = \bigcup_{(T \longrightarrow rhs) \in \mathcal{R}} \gamma(rhs)$$
$$\gamma(f(T_1, \ldots, T_n)) = \{f(t_1, \ldots, t_n) \mid t_i \in \gamma(T_i)\}$$

The types of interest are each defined by one grammar: each T_i is defined by a grammar $(T_i, \mathcal{T}_i, \mathcal{F}, \mathcal{R}_i)$, so that for any two types of interest T_1 and T_2 on $\mathcal{F}$, $\mathcal{T}_1 \cap \mathcal{T}_2 = \emptyset$. Sometimes, we will be interested in types defined by non-terminals of a grammar $(T, \mathcal{T}, \mathcal{F}, \mathcal{R})$ other than the distinguished non-terminal T. This is formalized by defining a type $T_i \in \mathcal{T}$ as the grammar

$$(T_i, \{T \in \mathcal{T} \mid T_i \overset{reach}{\longrightarrow}{}^*_{\mathcal{R}} T\}, \mathcal{F}, \{(T \longrightarrow rhs) \in \mathcal{R} \mid T_i \overset{reach}{\longrightarrow}{}^*_{\mathcal{R}} T\}) \qquad (1)$$

where all the non-terminals are renamed apart, $\overset{reach}{\longrightarrow}{}^*_{\mathcal{R}}$ is the reflexive and transitive closure of $\overset{reach}{\longrightarrow}_{\mathcal{R}}$ and

$$T_i \overset{reach}{\longrightarrow}_{\mathcal{R}} T_j \text{ iff } T_i \longrightarrow_{\mathcal{R}} T_j \text{ or } T_i \longrightarrow_{\mathcal{R}} f(\ldots, T_j, \ldots).$$

A grammar is in *normal form* if none of the right hand sides are non-terminals. A particular class of grammars are deterministic ones. A grammar is *deterministic* if it is in normal form and for each non-terminal T the function symbols are all distinct in the right hand sides of the productions for T.

Deterministic grammars are less expressive than non-deterministic ones. Deterministic grammars can only express sets of terms which are *tuple-distributive*; informally speaking, which are "closed under exchange of arguments". I.e., if the set contains two terms of the same functor, then it also contains terms with the same principal functor obtained by exchanging subterms of the previous two terms in the same argument positions. Basically, no dependencies between arguments of a term can be expressed with deterministic grammars.

Example 1. Consider the type T denoting the set $\{f(a, b), f(c, d)\}$, which is non-deterministic,

$$T \longrightarrow f(A, B) \;\; A \longrightarrow a \;\; C \longrightarrow c$$
$$T \longrightarrow f(C, D) \;\; B \longrightarrow b \;\; D \longrightarrow d$$

A deterministic type T' with a concretization which included $\gamma(T)$ would also have to include $\{f(c,b), f(a,d)\}$, that is,

$$T' \longrightarrow f(AC, BD) \;\; \begin{array}{ll} AC \longrightarrow a & BD \longrightarrow b \\ AC \longrightarrow c & BD \longrightarrow d \end{array}$$

To facilitate the presentation non-terminals with a single production will often be "inlined" and multiple right hand sides combined so that T above will be written $T \longrightarrow f(a,b) \mid f(c,d)$ and T' as

$$T' \longrightarrow f(AC, BD) \;\; AC \longrightarrow a \mid c \;\; BD \longrightarrow b \mid d$$

To be able to describe terms containing numbers and variables we introduce two distinguished symbols **num** and **any**, plus an additional $\perp$. The concretization of **num** is the set of all numbers, the concretization of **any** is the set of all terms (including variables), and the concretization of $\perp$ is the empty set of terms. These symbols are non-terminals but they are considered terminals to the effect of regarding a grammar as deterministic.

Let $\mathcal{G}$ be the set of all grammars, if T_1, T_2 belong to $\mathcal{G}$, the relation $T_1 \equiv T_2 \Leftrightarrow \gamma(T_1) = \gamma(T_2)$ is an equivalence relation. The quotient set $\mathcal{G}/\equiv$ is a complete lattice with top element **any** and bottom element $\perp$ based on the relation of *containment*, or type *inclusion*: for every $\overline{T_1}, \overline{T_2} \in \mathcal{G}/\equiv$, $\overline{T_1} \sqsubseteq \overline{T_2} \Leftrightarrow \gamma(T_1) \subseteq \gamma(T_2)$. We will denote $\overline{T_i}$ simply by T_i.

The least upper bound is given by type *union*, $(T_1 \sqcup T_2)$, and the greatest lower bound by type *intersection*, $(T_1 \sqcap T_2)$ [3]. It can be shown that intersection describes term unification:

$$t_1^* \subseteq \gamma(T_1) \wedge t_2^* \subseteq \gamma(T_2) \wedge t_1\theta = t_2\theta \Rightarrow (t_1\theta)^* \subseteq \gamma(T_1 \sqcap T_2)$$

where t^* denotes the set of ground terms which are instances of the term t.

3 Abstract Domain for Type Inference

In an abstract interpretation-based type analysis, a type is used as an abstract description of a set of terms. Given variables of interest $\{x_1, \ldots, x_n\}$, any substitution $\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ can be approximated by an *abstract substitution* $\{x_1 \leftarrow T_{x_1}, \ldots, x_n \leftarrow T_{x_n}\}$ where $t_i \in \gamma(T_{x_i})$ and each type $T_{x_i} \in \mathcal{G}/\equiv$. We will write abstract substitutions as tuples $\langle T_1, \ldots, T_n \rangle$, and sometimes also abbreviate a tuple simply as T^n.

Concretization is lifted up to abstract substitutions straightforwardly,

$$\gamma(\langle T_1, \ldots, T_n \rangle) = \{ \; \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\} \mid t_i \in \gamma(T_i) \; \}$$

as well as the equivalence relation $\equiv$. Additionally, we consider a distinguished abstract substitution $\perp$ as a representative of any $\langle T_1, \ldots, T_n \rangle$ such that there is $T_i = \perp$. Of course, $\gamma(\perp) = \emptyset$.

An ordering on the domain is obtained as the natural element-wise extension of the ordering on types:

$$\begin{array}{l} \bot \sqsubseteq T^n \\ \langle T_1, \ldots, T_n \rangle \not\sqsubseteq \bot \\ \langle T_1, \ldots, T_n \rangle \sqsubseteq \langle T'_1, \ldots, T'_n \rangle \iff \forall_{1 \leq i \leq n} T_i \sqsubseteq T'_i \end{array}$$

The domain is a lattice with bottom element $\bot$ and top element $\langle T_1, \ldots, T_n \rangle$ such that $T_1 = \ldots = T_n =$ **any**. The greatest lower bound and least upper bound domain operations are lifted also element-wise, as follows,

$$\begin{array}{c} \bot \sqcup T^n = T^n \sqcup \bot = T^n \\ \langle T_1, \ldots, T_n \rangle \sqcup \langle T'_1, \ldots, T'_n \rangle = \langle T_1 \sqcup T'_1, \ldots, T_n \sqcup T'_n \rangle \\ \bot \sqcap T^n = T^n \sqcap \bot = \bot \\ \langle T_1, \ldots, T_n \rangle \sqcap \langle T'_1, \ldots, T'_n \rangle = \langle T_1 \sqcap T'_1, \ldots, T_n \sqcap T'_n \rangle \end{array}$$

Using the adjoint α of γ as abstraction function, it can be shown that $(2^{\Theta}, \alpha, \Omega, \gamma)$ is a Galois insertion, where Θ is the domain of concrete substitutions and Ω that of abstract substitutions.

The following abstract unification operator can be shown to approximate the concrete one. Let $x = t$ be a concrete unification equation, with x a variable,t any term, and T^n the current abstract substitution, and let y_j, $j = 1, \ldots, m$ be the variables of t, the new abstract substitution is:

$$amgu(T^n, x = t) = T^n[T_x/T'_x, T_{y_1}/T'_{y_1}, \ldots, T_{y_m}/T'_{y_m}] \tag{2}$$

with each T replaced by T' in the tuple, $T'_x = T_x \sqcap t\mu$, $\mu = \{y_1 \leftarrow T_{y_1}, \ldots, y_m \leftarrow T_{y_m}\}$, and $solve(t, T'_x) = \{y_1 = T'_{y_1}, \ldots, y_m = T'_{y_m}\}$, a set of equations that define the types of the variables of a term $t \in \gamma(T'_x)$, obtained as:

$$solve(t, T) = \begin{cases} \{t = T\} & \text{if } t \text{ is a variable} \\ \bigcup_{T \longrightarrow f(T_1, \ldots, T_n)} \bigcup_{i=1,\ldots,n} solve(t_i, T_i) & \text{if } t \text{ is } f(t_1, \ldots, t_n) \end{cases}$$

In this abstract interpretation-based setting, analysis with a monotonic semantic function can be easily shown correct. However, it is not guaranteed to terminate, since Ω has infinite ascending chains. To guarantee termination, a widening operator is required.

Example 2. Consider the following program which defines the regular type lists of lists of numbers:

```
list_of_lists([]).
list_of_lists([L|Ls]):-
        num_list(L),
        list_of_lists(Ls).
```

```
num_list([]).
num_list([N|Xs]):-
        number(N),
        num_list(Xs).
```

For the argument of `num_list`, without a widening operator, an analysis would obtain the following first three approximations:

$$T_0 \longrightarrow [] \quad T_1 \longrightarrow [] \mid .(\mathbf{num}, T_0) \quad T_2 \longrightarrow [] \mid .(\mathbf{num}, T_1)$$

where each T_i represents a list of i numbers. Analysis will never terminate, since it would keep on obtaining a new type representing a list with one more number. A widening operator would be required that over-approximates some type T_l to something like

$$T_l \longrightarrow [] \mid .(\mathbf{num}, T_l)$$

which is the expected type, and allows termination of the analysis.

4 Widenings

The widening operation is required to guarantee that an analysis terminates when the abstract domain has infinite ascending chains as is the case of regular types.

Functor Widening. This is probably the simplest widening operator which still keeps information from the recursive structure of the program that "produces" the corresponding terms. The idea behind it is to create a type and a production for each functor symbol in the original type. All arguments of the function symbols are replaced with the new types [11].

Example 3. Consider predicate `list_of_lists` of Example 3.2, its argument should ideally have the following type:

$$T_{ll} \longrightarrow [] \mid .(T_l, T_{ll}) \quad T_l \longrightarrow [] \mid .(\mathbf{num}, T_l)$$

but the functor widening will yield

$$T \longrightarrow [] \mid \mathbf{num} \mid .(T, T)$$

Type Jungle Widening. A type jungle is a grammar where each functor always has the same arguments. It was originally proposed as a finite type domain [10], since in a domain where all grammars are of the type jungle class all ascending chains are finite. However, it can be used as a subdomain to provide a widening operator.

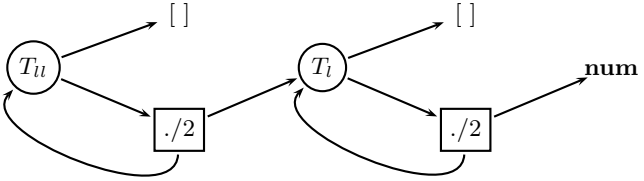
Example 4. Applying this widening to the previous type T_{ll}, the following will be obtained:

$$T \longrightarrow [] \mid .(T_1, T) \quad T_1 \longrightarrow [] \mid \mathbf{num} \mid .(T_1, T)$$

Note that this widening is strictly more precise than the functor widening. In the example, the new type captures the upper level of lists, but it loses precision when describing the type of the list elements. This is due to the restriction of forcing functors to always have the same arguments.

Shortening. A grammar can be seen as a graph where the nodes correspond to the non-terminals (or-nodes) and to the right hand sides of productions (and-nodes), and the edges correspond to the production relation or the relation between a functor and its arguments in a right hand side of a production. Given an or-node, its *principal functors* are the functors appearing in its children nodes.

Example 5. The type T_{ll} of the previous examples can be seen as the graph:



Gallagher and de Waal [6] defined a widening which avoids having two or-nodes, which have the same principal functors, connected by a path. If two such nodes exist, they are replaced by their least upper bound.

Example 6. In the above example graph, nodes T_{ll} and T_l have the same principal functors ([] and .) so that they are replaced, yielding:

$$T \longrightarrow [] \mid .(T_1, T) \quad T_1 \longrightarrow [] \mid \mathbf{num} \mid .(\mathbf{num}, T)$$

Note the precision improvement with respect to the result in the previous example. Note also that still the result is imprecise.

Restricted Shortening. Saglam and Gallagher [12] propose a more precise variant of the previous widening. Shortening is restricted so that two or-nodes T and T' which are connected by a path from T to T' and have the same principal functors are replaced only if $T' \sqsubseteq T$. If this is the case, only T' needs be replaced, since the least upper bound is T.

Example 7. Continuing previous examples, since nodes T_{ll} and T_l have the same principal functors but $T_l \not\sqsubseteq T_{ll}$, the widening operation will make no change. In this case, the most precise type is achieved.

Note, however, that restricted shortening does not guarantee termination in general (and thus, it is not, strictly speaking, a widening). There are cases in which analysis may not terminate using only this widening operator [11].

Depth Widening. Janssens and Bruynooghe [9] proposed a type analysis in which the widening effect is achieved by a "pruning" of the type depth up to a certain bound. A parameter k establishes the maximum number of occurrences of a functor in-depth in a type. The idea is similar to the well-known depth-k abstraction for term structure analysis. The resulting type analysis uses normal restricted type graphs, which are basically deterministic types satisfying the depth limit. Obviously, precision of this analysis depends on the value of the parameter k.

Example 8. The widening of our previous type T_{ll} with k=1 will yield the same result than the functor widening (Example 4.3), whereas with k=2 will yield the same result as restricted shortening (Example 4.7).

Topological Clash Widening. Van Hentenryck et al. [13] proposed the first widening operator that takes into account two consecutive approximations to the type being inferred. After merging the two —i.e., calculating their least upper bound, the result is compared with the previous approximation to try to "guess" where the type is growing. This is done by locating *topological clashes*: functors that differ or appear at different depth in each type graph. The clashes are resolved by replacing them with the recently calculated least upper bound.

Example 9. Consider the program:

```
sorted([]).
sorted([_X]).
sorted([X,Y|L]):- X =< Y, sorted([Y|L]).
```

and the moment during analysis when the final widening is performed. The resulting type for the argument of `sorted/1` is the one on the left below for the first two clauses, and the one on the right for the last one:

$$T_0 \longrightarrow [] \mid .(\mathbf{any}, []) \quad \begin{array}{l} T_1 \longrightarrow .(\mathbf{num}, .(\mathbf{num}, T_l)) \\ T_l \longrightarrow [] \mid .(\mathbf{num}, T_l) \end{array}$$

Their least upper bound is T_u on the left below, which exhibits a clash with T_0 in the second argument of functor $./2$. Thus, the result of widening is T_s:

$$T_u \longrightarrow [] \mid .(\mathbf{any}, T_l) \quad T_s \longrightarrow [] \mid .(\mathbf{any}, T_s)$$

All widening operators are based on locating recursive structures in the type definitions where there are nodes with the same functors. This may originate an unnecessary loss of precision, since the widening may impose a recursive structure on the resulting type in argument positions where the concrete program is in fact not recursive. In the following section we present a new widening operator that tries to remedy this problem.

5 Structural Type Widening

In this section we define an extended domain for type analysis which incorporates a widening operator aimed at improving the precision of the analysis. The domain is defined so as to keep track of information on the program structure, so that recursion on the types produced by the analysis is imposed by the widening operator only in the cases where it corresponds to a recursive structure in the program being analyzed. To this end, type names will be used.

A *type name* is roughly a (distinguished) non-terminal that represents a type produced during the analysis. Type names are created for each variable in each argument of each variant of each program atom for each predicate (note how this is different from, for example, set-based analyses [1], where variants are not taken into account).

Type names provide information on how types are being formed from other types during analysis. This makes it possible to precisely identify places where

to impose recursion on the types: in a subterm of the type which happens to refer to the name of that type. To this end, type names contain references to the position of its constituent types. To determine positions, selectors are used, as defined below.

Definition 1 (selector). *Define t/s, the subterm of a concrete term t referenced by a* selector *s, inductively as follows. The empty selector ϵ refers to the term t, that is, $t/\epsilon = t$. If $t/s = t'$, t' is a compound term $f(t_1, \ldots, t_i, \ldots, t_n)$ (where f is an n-ary function symbol) then $t/s \cdot (f.i) = t_i$, $1 \leq i \leq n$.*

For every two selectors s, p, if $t/s = t'$ and if t'/p exists then $t/s \cdot p = t'/p$. The initial ϵ of a non-empty selector will often be omitted, so $\epsilon \cdot p$ will be written simply as p.

We define a set of type names $\mathcal{N}$ such that $\mathcal{N} \cap \mathcal{G} = \emptyset$ and a set $2^{\mathcal{N} \times \mathcal{G}}$ of relations $\mathcal{X} \in 2^{\mathcal{N} \times \mathcal{G}}$ between type names and types, of the form $\mathcal{X} \subseteq \mathcal{N} \times \mathcal{G}$.

Definition 2 (label). *Let $\mathcal{X}$ a relation between type names and types. Given a type name N, a selector s, and a type name N', a tuple $\langle s, N' \rangle$ is a* label *of N iff $(N, T) \in \mathcal{X}$, $(N', T') \in \mathcal{X}$, and $T' \sqsubseteq T/s$.*

Labels of a type name N indicate subterms of the type T defining N where other type names occur.

Example 10. Let a relation $\mathcal{X}$ such that $\{(A, T_1), (B, T_2)\} \subseteq \mathcal{X}$, and let grammars $(T_1, \mathcal{T}_1, \mathcal{F}, \mathcal{R}_1)$ and $(T_2, \mathcal{T}_2, \mathcal{F}, \mathcal{R}_2)$, such that the only rule for T_1 is $(T_1 \longrightarrow f(b)) \in \mathcal{R}_1$ and $(T_2 \longrightarrow g(c, T_3)) \in \mathcal{R}_2$, $(T_3 \longrightarrow b \mid f(b)) \in \mathcal{R}_2$. Consider a label $\langle (g.2), A \rangle$ of B. We have that $T_1 \sqsubseteq T_2/(g.2) = T_3$.

Definition 3 (type descriptor). *Let $\mathcal{G}$ a set of types (regular term grammars), $\mathcal{N}$ a set of type names, and $\mathcal{X} \subseteq \mathcal{N} \times \mathcal{G}$. A* type descriptor *is a tuple (N, E, T) where $N \in \mathcal{N}$, $T \in \mathcal{G}$, $(N, T) \in \mathcal{X}$, and E is a set of labels of N.*

In the new domain, type descriptors will be used instead of types. Let $\mathcal{D}$ be the set of all type descriptors from given sets of types $\mathcal{G}$ and of type names $\mathcal{N}$. Concretization is defined as $\gamma((N, E, T)) = \gamma(T)$. The domain ordering and operations on $\mathcal{D}$ are the same as on $\mathcal{G}$ except for type names. In this case, they have to take into account the possible labels of the type name.[1]

Inclusion. $(N_1, E_1, T_1) \sqsubseteq (N_2, E_2, T_2) \Leftrightarrow T_1 \sqsubseteq T_2 \wedge E_1 \subseteq E_2$.

Union. $(N, E, T) = (N_1, E_1, T_1) \sqcup (N_2, E_2, T_2) \Leftrightarrow T = T_1 \sqcup T_2 \wedge E = E_1 \cup E_2$.

Intersection. $(N, E, T) = (N_1, E_1, T_1) \sqcap (N_2, E_2, T_2) \Leftrightarrow T = T_1 \sqcap T_2 \wedge E = E_1 \cup E_2$.

[1] Note that these operations do not manipulate the type names: they are assigned independently during analysis. In particular, the name N of the type resulting from union and intersection is always a new name.

Again, we may be interested in types defined by non-terminals other than the distinguished non-terminal T of a grammar $(T, \mathcal{T}, \mathcal{F}, \mathcal{R})$. A type descriptor (N_i, E_i, T_i), where $T_i \in \mathcal{T}$, is formally defined from (N, E, T) as follows: T_i is the grammar of Equation 1, N_i is a new type name, and

$$E_i = \{\langle p, N' \rangle \mid \langle s \cdot p, N' \rangle \in E \wedge T/s = T_i\}.$$

Abstract substitutions for variables of interest $\{x_1, \ldots, x_n\}$ are now defined as tuples of the form $\langle (N_1, E_1, T_{x_1}), \ldots, (N_n, E_n, T_{x_n}) \rangle$. Concretization and the domain ordering and operations are lifted to abstract substitutions element-wise, in the same way as in Section 3, including the widening operator defined below. If now Ω is the domain of type descriptors, it can be shown that $(2^{\Theta}, \alpha, \Omega, \gamma)$ is a Galois insertion, where α is the adjoin of γ. Abstract unification is defined as in Equation 2, but using type descriptors instead of types. During unification, all type names in the "input" abstract substitution T^n to *amgu* are preserved; in the labels, the selectors for those names are changed so as to refer to the resulting type graph instead of to that of T^n.

Definition 4 (structural widening). *The widening between an approximation T_2 to type name N and a previous approximation T_1 to N is $(N, E_1, T_1) \bigtriangledown (N, E_2, T_2) = (N, E_1 \cup E_2, T)$, such that T is defined by $(T, \mathcal{T}, \mathcal{F}, \mathcal{R})$ where $\mathcal{T} = \{T_i \mid T \longrightarrow^*_{\mathcal{R}} T_i\}$, and $\mathcal{R}$ is obtained by the following algorithm:*

$T' := T_1 \sqcup T_2$ defined by $(T', \mathcal{T}', \mathcal{F}, \mathcal{R}')$
$\mathcal{S} := \{s \mid (s, N) \in E_1 \cup E_2\}$
$Seen := \emptyset$
`for each` $(T' \longrightarrow f(A_1, \ldots, A_n)) \in \mathcal{R}'$ `add to` $\mathcal{R}$ `production`
  $T \longrightarrow f(\texttt{widen}(A_1, \mathcal{R}', (f.1)), \ldots, \texttt{widen}(A_n, \mathcal{R}', (f.n)))$

$\texttt{widen}(N, \mathcal{R}', Sel)$:
  `if` $N =$ **any** `return` **any**
  `if** $\exists M \langle N, M \rangle \in Seen$ `return` M
  `let` M `a new non-terminal`
  $Seen$ `:=` $Seen \cup \{\langle N, M \rangle\}$
  `for each` $(N \longrightarrow f(A_1, \ldots, A_n)) \in \mathcal{R}'$ `add to` $\mathcal{R}$ `production`
    $M \longrightarrow f(\texttt{widen}(A_1, \mathcal{R}', Sel \cdot (f.1)), \ldots, \texttt{widen}(A_n, \mathcal{R}', Sel \cdot (f.n)))$
  `if` Sel $\in \mathcal{S}$ `then`
    `add to` $\mathcal{R}$ `production` $M \longrightarrow T$
  `return` M

Structural widening basically identifies subterms of the new type $T_1 \sqcup T_2$ where a reference to the type N being widened appears, and makes this "self-reference" explicit in the definition of the new type. Note that the widening operation starts with the least upper bound and, basically, adds new grammar rules to that type. Therefore, the result is always a correct approximation of such an upper bound. This justifies its correctness. Moreover, this approach based on type names is potentially more precise than any of the previous widening operators discussed, as the following examples show:

Example 11. Consider program `sorted` in Example 4.9. A top-down analysis with topological clash was roughly described there. Let us now look at analysis using restricted shortening. The resulting type happens to be the same one.

Analysis of program atom `sorted([Y|L])` approximates variable `Y` always as **num**, both in the calls and in the successes. The first two success approximations for variable `L` are $[]$ and $.(\mathbf{num}, [])$. Their lub (and widening) is:

$$T_1 \longrightarrow [] \mid .(\mathbf{num}, [])$$

The next approximation to the type of `L` is $.(\mathbf{num}, T_1)$. Its lub with T_1 is $T_2 \longrightarrow [] \mid .(\mathbf{num}, T_1)$, and since T_2 and T_1 have the same functors, and T_1 is included in T_2, the widening of T_2 is:

$$T_3 \longrightarrow [] \mid .(\mathbf{num}, T_3)$$

i.e., list of numbers. The next approximation to the type of `L` is $.(\mathbf{num}, T_3)$ (i.e., a list with at least one number). It is included in T_3, so fixpoint is reached.

The success of principal goal `sorted(X)` is approximated after analyzing the two non-recursive clauses by $T_4 \longrightarrow [] \mid .(\mathbf{any}, [])$. Analysis of the third clause yields $.(\mathbf{num}, .(\mathbf{num}, T_3))$. Its lub with T_4 is $T_5 \longrightarrow [] \mid .(\mathbf{any}, T_3)$. The widening of T_5 finds that T_5 and T_3 have the same functors and $T_3 \sqsubseteq T_5$, since $\mathbf{num} \sqsubseteq \mathbf{any}$. Thus, the result of widening is:

$$T_6 \longrightarrow [] \mid .(\mathbf{any}, T_6)$$

i.e., list of terms. This is the final result after one more iteration. Note that the information about successes where the tail of lists of length greater than one is a list of numbers is lost.

Let us now consider structural widening. Analysis of atom `sorted([Y|L])` always approximates the type of `Y` by $(N_{13}, \emptyset, \mathbf{num})$. For variable `L` the two first approximations are $(N_{14}, \emptyset, [])$ and $(N_{14}, E_{14}, .(\mathbf{num}, []))$, where the set of labels is $E_{14} = \{\ (\text{'.'}.1, N_{13}),\ (\text{'.'}.2, N_{14})\ \}$. The result of widening is (N_{14}, E_{14}, T_1) where T_1 is defined as:

$$T_1 \longrightarrow [] \mid .(\mathbf{num}, T_1)$$

i.e., list of numbers. This is the final result after one more iteration.

The success of principal goal `sorted(X)` is approximated after analyzing the two non-recursive clauses by $(N_3, \emptyset, T_2)$ where $T_2 \longrightarrow [] \mid .(\mathbf{any}, [])$. Analysis of the third clause yields $(N_3, E_3, .(\mathbf{num}, .(\mathbf{num}, T_1)))$, where

$$E_3 = \{\ (\text{'.'}.2 \cdot \text{'.'}.1, N_{13}),\ (\text{'.'}.2 \cdot \text{'.'}.2, N_{14})\ \}$$

Its widening with the previous approximation T_2 is (N_3, E_3, T_3), where

$$T_3 \longrightarrow [] \mid .(\mathbf{any}, T_1)$$

which amounts to their lub, since the widening operator does not produce any change, because N_3 is not among its own labels. Therefore, the final result, after one more iteration, is T_3, where indeed lists of length greater than one have a tail which is a list of numbers.

However, structural widening does not guarantee termination. It is effective as long as the new approximation is built from the previous approximation of the type being inferred. This case is identified, in essence, by locating a reference to the type name of the previous approximation within the definition of the new one. However, there are contrived cases in which a type is constructed during analysis which loses the reference to the previous approximation. In these cases, a more restrictive widening has to be applied to guarantee termination.

Example 12. Consider the program:

```
main:- p(a).      p(a).                      q(a,f(a)).
                  p(X):- q(X,Y), p(Y).       q(f(Z),f(L)):- q(Z,L).
```

The calling substitution for atom `p(Y)` is the sequence

$$T_1 \longrightarrow f(a) \quad T_2 \longrightarrow f(f(a)) \quad T_3 \longrightarrow f(f(f(a))) \quad \ldots$$

whereas the type $T \longrightarrow f(a) \mid f(T)$ correctly describes such calls. However, the analysis is not able to infer such a type.

The problem in the above example is that none of the approximations T_i contains a reference to the previous approximation. This is originated in the program fact for predicate `q/2` which causes the loss of the reference to the previous approximation because of the double occurrence of constant `a`.

In our analysis, termination is guaranteed by a bound on the number of times the widening operation can be applied to a type name. A counter is associated to each type name, so that when the bound is reached a more restrictive widening that guarantees termination is applied.

6 Analysis Results

We have implemented analyses based on most of the widenings discussed in this paper, including structural widening. The implementation is in Prolog and has been incorporated to the CiaoPP system [8,7], which uses the top-down analysis algorithm of PLAI. The analysis of [6], based on regular approximations, which uses a bottom-up algorithm, is also incorporated into the system. This analysis uses shortening. We want to compare the top-down and bottom-up approaches with the same widening and similar implementation technology,[2] as well as the precision and efficiency, within the same analysis framework, of the widening operators previously discussed.

We have used two sets of benchmark programs: the one used in the PLAI framework and that used in the GAIA [2] framework. A summary of the benchmarking follows. The analysis times in miliseconds are shown in Table 1. The first column (`rul`) is for the regular approximation analysis and the other three for the PLAI-based analyses: column `short` for shortening, column `clash` for topological clash, and column `struct` for structural widening.

[2] Similar in the programming technique. Of course, the regular approximation method is rather different from the method of program interpretation on an abstract domain: Evaluating this difference is part of the aim of the comparison.

Table 1. Timing results

Program	`rul`	`short`	`clash`	`struct`
aiakl	568	469	529	900
bid	1480	2209	2529	4730
boyer	3450	3890	4989	9629
browse	758	380	389	539
cs_o	3840	1889	2689	2580
cs_r	18549	10720	24479	19560
disj_r	4468	1819	6399	2440
gabriel	1549	1430	1870	1760
grammar	330	160	160	190
hanoiapp	620	719	1889	1150
kalah_r	1520	79	79	89
mmatrix	310	190	209	119
occur	380	219	330	289
palin	590	840	980	850
pg	839	2020	2980	3990
plan	1138	819	960	1009
progeom	979	1840	2530	3640
qsort	310	590	659	680
qsortapp	369	1000	2898	1210
queens	329	179	190	180
query	720	360	370	410
serialize	478	810	969	899
witt	2929	4890	1399	1169
zebra	560	3490	14958	12830

Table 2 shows results in terms of precision. The precision of `struct` is never improved by any of the others. The improved precision of `struct` has been measured as follows. The left subcolumns under `rul`, `short`, and `clash` show the number of types with a more precise definition inferred by `struct`. The right subcolumns show the number of types where the previous ones appear (and are thus, also, more precise). The former are types directly inferred from program predicates; the latter are types which are defined from the former, due to the data flow in the program.

The following conclusions can be drawn from the tables. First, the regular approximation approach seems to behave better in terms of efficiency than the program interpretation approach, at least for the bigger programs. This conclusion, however, has to be taken with some care, since the current implementation of `rul` performs some caching of the type grammars that the PLAI-based analysis does not. This should be subject of a more thorough evaluation, which is out of the scope of this paper. The fact that it improves in bigger programs seems to suggest that the effect of this caching is most surely not negligible.

Regarding the analyses based on program interpretation, it can be concluded that the better the precision the worse the efficiency: `short` takes less than

Table 2. Precision results

Program	`rul`		`short`		`clash`	
aiakl	1	1	1	1		
bid	9	12	9	12		
cs_o	4	18	4	18	2	9
cs_r	4	28	4	28	2	19
disj_r	6	13	6	13		
mmatrix	2	2	2	2		
occur	1	1	1	1		
palin	2	4	2	4		
pg	1	1	1	1		
qsort	1	1	1	1		
serialize	2	4	2	4		
zebra	3	3	3	3	1	1

`clash`, and this one takes less than `struct`; this one is more precise than `clash`, which is more precise than `short`. This conclusion seems evident at first sight, but it is not: in analysis, an improvement in precision can very well trigger an improvement in efficiency. This can also be seen in the tables in some cases, the most significant probably being `zebra`. Overall, one can arguably conclude that the efficiency loss found is not a high price in exchange for the gain in precision.

We have also carried out another test. For practical purposes, the CiaoPP system includes a back-end to the analysis that simplifies the types inferred, in the sense that equivalent types are identified, so that they are then reduced to a single type. This facilitates the interpretation of the output. It is the case that the structural widening includes certain amount of type simplification, so that the analysis creates less different types which are in fact equivalent. For this reason, we have included the same tests as above, but adding now the times taken in the back-end simplification phase.

The times including the simplification are shown in table 3. The columns read as before. It can be seen that in this case structural widening outperforms all of the other analyses, except, in some cases, `rul`.

It also can be observed that `rul` behaves usually better than `short` also when simplification is included. This seems to suggest that incorporating our widening into the regular approximation approach would probably give the best results in practice.[3]

7 Conclusions

We have presented a new widening operator on regular types within an abstract interpretation-based characterization of type inference. The idea behind it is similar to set-based analyses [4,1] in that we assign and fix type names, but it

[3] This, however, may not be trivial. It is subject for future work.

Table 3. Timing results (including simplification)

Program	`rul`	`short`	`clash`	`struct`
aiakl	697	3009	3738	1409
bid	2899	31278	35949	15259
boyer	19620	201169	206917	92117
browse	987	2848	2987	1698
cs_o	11958	17389	32959	4878
cs_r	50760	303430	238788	30169
disj_r	6508	18598	26077	6408
gabriel	2098	13388	22379	5208
grammar	759	3169	3169	1279
hanoiapp	840	3988	13738	3378
kalah_r	2069	1187	1188	888
mmatrix	757	1769	2078	488
occur	530	1647	2628	767
palin	997	8520	11878	2180
pg	1349	15380	22870	7370
plan	1587	6167	6559	2288
progeom	1358	12800	17598	6679
qsort	520	3439	4168	1409
qsortapp	569	7789	9669	2900
queens	457	1128	1138	429
query	1627	22458	22788	11818
serialize	937	8429	11957	2217
witt	3438	188419	42699	25709
zebra	717	55100	189949	44540

is applied here with more generality. The most comparable aproach among the set-based analyses would be [5]. It can be seen as a generalization of the idea of "guessing" the growth of the types during analysis which is behind [13]. Instead of guessing, our technique determines exactly where the type is growing. The resulting widening operator has been presented on deterministic regular types. However, its extension to non-deterministic regular types should be straightforward.

Our operator is more precise than previous approaches, but it is still efficient. This has been shown with (preliminary) practical results. However, it does not guarantee termination. We are currently working on the non-termination problem. A moded type domain will help in this. The idea is to enhance abstract unification so that it is able to identify the "transference" of type names from the input to the output types, so that the names are not dropped. This will remedy the problem of Example 12 and, hopefully, allow us to prove termination of analyses with the proposed widening operator.

Finally, this work has revealed two issues that may be worth investigating for practical purposes: the impact on the efficiency of analysis of the different

implementation techniques for different analysis methods, on one hand, and of the simplification of types, on the other hand.

Acknowledgements. We would like to thank John Gallagher for very useful discussions, and Pedro López for his help with the implementation and the availability of his library on type manipulation. This work has been partially supported by Spanish MCYT project EDIPIA TIC99-1151 and by EU funded projects AMOS IST-2001-34717 and ASAP IST-2001-38059.

References

1. W. Charatonik, A. Podelski, and J.-M. Talbot. Paths vs. Trees in Set-based Program Analysis. In *Principles of Programming Languages*, pages 330–338. ACM Press, January 2000.
2. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.
3. P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.
4. T. Früwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic Programs as Types for Logic Programs. In *Proc. LICS'91*, pages 300–309, 1991.
5. J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Fourth International Symposium on Practical Aspects of Declarative Languages*, LNCS, pages 243–261. Springer-Verlag, January 2002.
6. J.P. Gallagher and D.A. de Waal. Fast and Precise Regular Approximations of Logic Programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.
7. M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The Ciao Logic Programming Environment. In *International Conference on Computational Logic, CL2000*, July 2000.
8. M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.
9. G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.
10. T. Lindgren and P. Mildner. The Impact of Structure Analysis on Prolog Compilation. Technical Report 140, Computing Science Dept., Uppsala University, 1997.
11. P. Mildner. *Type Domains for Abstract Interpretation: A Critical Study.* PhD thesis, Computing Science Department - Uppsala University, 1999.
12. H. Saglam and J. Gallagher. Approximating Logic Programs Using Types and Regular Descriptions. Technical Report CSTR-94-19, Department of Computer Science, University of Bristol, Bristol BS8 1TR, 1994.
13. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type Analysis of Prolog Using Type Graphs. *Journal of Logic Programming*, 22(3):179–209, 1995.

A Few Graph-Based Relational Numerical Abstract Domains*

Antoine Miné

École Normale Supérieure de Paris, France,
mine@di.ens.fr,
http://www.di.ens.fr/~mine

Abstract. This article presents the systematic design of a class of relational numerical abstract domains from non-relational ones. Constructed domains represent sets of invariants of the form $(v_j - v_i \in C)$, where v_j and v_i are two variables, and C lives in an abstraction of $\mathcal{P}(\mathbb{Z})$, $\mathcal{P}(\mathbb{Q})$, or $\mathcal{P}(\mathbb{R})$. We will call this family of domains *weakly relational domains*. The underlying concept allowing this construction is an extension of potential graphs and shortest-path closure algorithms in exotic-like algebras.
Example constructions are given in order to retrieve well-known domains as well as new ones. Such domains can then be used in the Abstract Interpretation framework in order to design various static analyses. A major benefit of this construction is its modularity, allowing to quickly implement new abstract domains from existing ones.

1 Introduction

Proving the correctness of a program is essential, especially for critical and embedded applications (such as planes, rockets, and so on). Among several correctness criteria, one should ensure that a program can never perform a run-time error (divide by zero, overflow, etc.). A classical method consists in finding a *safety invariant* before each dangerous operation in the program, and checking that the invariant implies the good behavior of the subsequent operation. Because this task is to be performed on the whole program—containing maybe tens of thousands of lines—and must be repeated after even the slightest code modification, we need a purely automatic static analysis approach.

Discovering the tightest invariants of a program cannot be fully mechanized in general, so we have to find some kind of sound approximation. By sound, we mean that the analysis should find an over-approximation of the real invariant. We will always discover *all* bugs in a program. However, we may find false alarms.

* This work was supported in part by the RTD project IST-1999-20527 "DAEDALUS" of the European IST FP5 program.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 117–132, 2002.
© Springer-Verlag Berlin Heidelberg 2002

2 Previous Work

We will work in the well-known *Abstract Interpretation* framework, proposed by Cousot and Cousot in [6,7], which allows us to easily describe sound and computable semantics approximations.

2.1 Numerical Abstract Domains

The crux of the method is to design a so-called *abstract domain*, that is to say, a practical representation of the invariants we want to study, together with a fixed set of operators and transfer functions (union, intersection, widening, assignment, guard, etc.) used to mimic the semantics of the programming language.

We will consider here *numerical abstract domains*. Given the set $\mathcal{V}$ of the numerical variables of a program with value in the set $\mathbb{I}$ (that can be $\mathbb{Z}$, $\mathbb{Q}$ or $\mathbb{R}$), a numerical abstract domain will represent and manipulate subsets of $\mathcal{V} \mapsto \mathbb{I}$. Well-known *non-relational domains* include the *interval* domain [5] (describing invariants of the form $v_i \in [c_1, c_2]$), the *constant propagation* domain ($v_i = c$), and the *congruence* domain [14] ($v_i \in a\mathbb{Z} + b$). Well-known *relational domains* include the *polyhedron* domain [9] ($\alpha_1 v_1 + \cdots + \alpha_n v_n \leq c$), the *linear equality* domain [18] ($\alpha_1 v_1 + \cdots + \alpha_n v_n = c$), and the *linear congruence equality* domain [15] ($\alpha_1 v_1 + \cdots + \alpha_n v_n \equiv a\ [b]$).

Non-relational domains are fast, but suffer from poor precision: they cannot encode relations between variables of a program. Relational domains are much more precise, but also very costly. Consider, for example, the simple program of Figure 1 that simulates many random walks and stores the hits in an array. Our goal is to discover that, at program point ($\bullet$), x is in the set $\{-5, -3, -1, 1, 3, 5\}$ of allowed indices for hit, so that the instruction $\mathsf{hit[x]}$++ is correct. The invariants found at ($\bullet$) by several methods are shown in Figure 2. Remark that, even if the desired invariant is a simple combination of an interval and a congruence relation, all non-relational analyses fail to discover it because they cannot infer the relationship between x and i at program point ($\bigstar$). It is often the case that, in order to find a given invariant at the end of a loop, one must be able to express invariants of a more complex form inside the loop. In this example, the desired result can be obtained by using relational analyses, as shown in Figure 2.

2.2 Graph-Based Algorithms

Pratt remarked in [25] that the satisfiability of a set of constraints of the form ($x - y \leq c$) can be efficiently tested in $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$ by looking at the simple loops of a directed weighted graph—so-called *potential graph*. Shostak then extended in [26] this graph-based algorithm to the satisfiability of constraints of the form ($\alpha x + \beta y \leq c$), in $\mathbb{Q}$ or $\mathbb{R}$. Harvey and Stuckey proved in [17] that Shostak's algorithm can be used to check satisfiability of constraints of the form ($\pm x \pm y \leq c$) in $\mathbb{Z}$. These approaches focus only on satisfiability and do not address the problem of manipulating constraint sets.

```
hit: array {-5,-3,-1,1,3,5} → int;
for k=1 to 1000 do
  x=0;
  for i=1 to 5 do
    (★) if random() then x++; else x--;
  done;
  (•) hit[x]++;
done
```

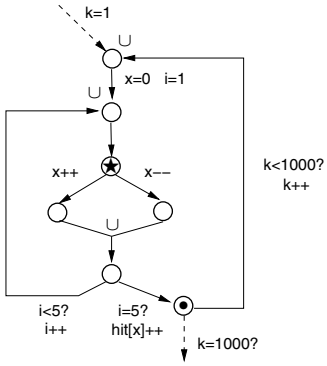


Fig. 1. A simple random walk program, and its control flow graph.

	Interval	Congruence	Polyhedron	Congruence equality
(★)	$i \in [1,5]$	—	$i \in [1,5],\ -x \leq i-1 \leq x$	$x+i \equiv 1\ [2]$
(•)	—	—	$x \in [-5,5]$	$x \equiv 1\ [2]$

Fig. 2. Invariants discovered for the program in Figure 1, at program points (•) and (★), using several non-relational (left) and relational (right) analyses.

Using Pratt's remark, the model-checking community developed a structure called *Difference-Bound Matrix* (DBM, for short) and algorithms based on *shortest-path closure* of weighted graphs to represent and manipulate constraints of the form $(x - y \leq c)$ and $(x \leq c)$. DBMs are used to model-check timed-automata. In [27], Toman and Chomicki introduced *periodicity graphs* that manipulate constraints of the form $(x \equiv y + c\ [k])$, and apply this to constraint logic programming and database query.

Unlike model-checking and constraint programming, we would like to analyze generic programs, and not simply systems closed under restricted constraint forms—such as timed automata, or database query languages. Our methodology is first to choose an invariant form, and then to design a fully-featured abstract domain (including guard and assignment transfer functions, as well as a widening operator) allowing to discover invariants of this form on any program, using maybe coarse over-approximations for those semantics functions that cannot be represented exactly using the chosen invariant form.

In [22], we already presented a DBM-based abstract domain allowing to discover invariants of the form $(x-y \leq c)$ and $(x \leq c)$. In [23], we presented a slight extension, called the *octagon abstract domain*, allowing to discover invariants of the form $(\pm x \pm y \leq c)$.

2.3 Our Contribution

Our goal is to propose a new family of numerical abstract domains, based on shortest-path closure algorithms, that allows to discover invariants of the form $(x-y \in C)$, where C lives in a non-relational domain. This family generalizes the DBM-based abstract domain of [22] and allows us to build new domains, such as the *zone congruence domain* that discovers invariants of the form $(x \equiv y+c\ [k])$. Such relational domains are between, in term of cost and precision, non-relational domains and classical relational domains—such as polyhedron or congruence equality. Thus, we will call these domains *weakly relational domains*.

We claim that such domains are useful as they give, on the example of Figure 1, almost the same result as the relational analyses, for a smaller cost. Do not be fooled by the simplicity of this example program; the abstract interpretation framework allows the design of complex inter-procedural analyzes [1] adapted to real-life programming languages. A numerical abstract domain is just a brick in the design of an analysis; it can be plugged in many existing analyses, such as pointer [10], string cleanness [11], termination analyses [3], analyses of mobility [12], probabilistic programs [24], abstraction of tree-based semantics [21], etc.

The paper is organized as follows. Section 3 reformulates the construction of non-relational numerical abstract domains using the concept of *basis*. Section 4 explains our generic construction of weakly relational domains and applies it in order to retrieve the zone domain and build new abstract domains. Section 5 provides a few applications and ideas for improvement. We conclude in Section 6.

3 Bases and Non-relational Numerical Abstract Domains

This sections first recalls the concept of numerical abstract domain. We introduce the new concept of *basis* and show how it can be used to retrieve standard non-relational domains. Introducing such a concept only for this purpose would be formalism for the sake of formalism. However, we will show in the next section how to use this concept to build our weakly relational domains. Hence, bases are the common denominator between classical non-relational domains and our weakly relational domain family.

From the implementation point of view, bases are modules sharing a common signature, and we propose one functor for building non-relational domains from this signature, and one functor for building weakly relational domains. From the mathematical point of view, this approach makes our proofs modular and easier to handle.

3.1 Semantics and Abstract Domains

Let P be a procedure-free, pointer-free program such as the one in Figure 1. Let $\mathcal{V} = \{v_0, \ldots, v_{N-1}\}$ be the set of its numerical variables, with values in the set $\mathbb{I}$ (that can be $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$). We attach to each node of the control flow graph of

P a set of *environments* $\mathbf{e}^\natural \in \mathcal{D}^\natural \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{I}^N)$ that maps each variable to its value. The information is propagated using the following equations:

- *guards*, corresponding to tests in the initial program, filter the environment: $\mathbf{e}^\natural_{(\texttt{expr ?})} \stackrel{\text{def}}{=} \{\ (x_0, \ldots, x_{N-1}) \in \mathbf{e}^\natural \mid \texttt{expr}(x_0, \ldots, x_{N-1}) \text{ holds}\ \}$;
- *assignments* change the value of one variable: $\mathbf{e}^\natural_{(v_i \leftarrow \texttt{expr})} \stackrel{\text{def}}{=} \{\ (x_0, \ldots, \texttt{expr}(x_0, \ldots, x_{N-1}), \ldots) \mid (x_0, \ldots, x_i, \ldots) \in \mathbf{e}^\natural\ \}$;
- *union* $\cup$ collects environments at control flow joins.

Because of loop constructs, the control flow graph contains loops and the system of equations described above is recursive. Classical safety semantics consider the *least fixpoint* solution.

This semantics is not decidable in general. One thus constructs an *abstract domain* [6] which is a computer-representable partially ordered set $(\mathcal{D}, \preceq)$ connected to $(\mathcal{D}^\natural, \subseteq)$ by a monotonic concretization function Γ. Guard, assignment, and union operators have *sound over-approximations* in $\mathcal{D}$, that is to say:

$$\begin{cases} \Gamma(\mathbf{e}_{(\texttt{expr ?})}) & \supseteq\ (\Gamma(\mathbf{e}))^\natural_{(\texttt{expr ?})}; \\ \Gamma(\mathbf{e}_{(v_i \leftarrow \texttt{expr})}) & \supseteq\ (\Gamma(\mathbf{e}))^\natural_{(v_i \leftarrow \texttt{expr})}; \\ \Gamma(\mathbf{e} \sqcup \mathbf{f}) & \supseteq\ \Gamma(\mathbf{e}) \cup \Gamma(\mathbf{f})\ . \end{cases}$$

Unlike classical data-flow analysis, $\mathcal{D}$ can have an infinite height, so one needs a *widening operator* [6] to compute, in finite-time, an over-approximation of least fixpoints. The widening operator $\triangledown$ should have the following properties [6]:

Definition 1. Widening.

1. $\forall x, y \in \mathcal{C},\ x \preceq x \triangledown y,$ *and* $y \preceq x \triangledown y$.
2. *For every increasing sequence* $(y_n)_{n\in\mathbb{N}}$, *the sequence* $(x_n)_{n\in\mathbb{N}}$ *defined by* $\begin{cases} x_0 & = y_0, \\ x_{n+1} & = x_n \triangledown y_n, \end{cases}$ *is ultimately stationary. (Ascending chain condition.)*

$\Diamond$

The least fixpoint $\text{lfp}_\perp F$ of an abstract operator F is replaced by the limit of the *stationary* sequence $X_0 = \perp$, $X_{i+1} = X_i \triangledown F(X_i)$—see [2] for more information on when and how to use widenings.

It is a major result of abstract interpretation that, when computing in the abstract domain with widenings, one obtains, in finite time, a sound over-approximation of the initial semantics.

3.2 Bases

We call *basis* a structure that represents and manipulates subsets of $\mathbb{I}$ in a way suitable to build a *non-relational* abstract domain. Such bases will be then used in the following section to build our family of *relational* domains. It is given by:

Definition 2. Basis.

1. *A computer-representable set* $\mathcal{C}$ *with partial order* $\sqsubseteq$ *and least element* $\bot$.
2. *A strict, monotonic, injective concretization* $\gamma : \mathcal{C} \hookrightarrow \mathcal{P}(\mathbb{I})$.
3. *Each element* $C \subseteq \mathbb{I}$ *has an over-approximation* $C^\sharp \in \mathcal{C}$: $\gamma(C^\sharp) \supseteq C$.
4. *There exists an over-approximation* $\sqcap$ *for the intersection:*
$$\gamma(C_1^\sharp \sqcap C_2^\sharp) \supseteq \gamma(C_1^\sharp) \cap \gamma(C_2^\sharp) \ .$$
5. *There exists an upper bound* $\sqcup$: $\quad C_1^\sharp, C_2^\sharp \sqsubseteq C_1^\sharp \sqcup C_2^\sharp$.
6. *Each* k*-ary arithmetic expression* $\boldsymbol{expr}_k(c_1, \ldots, c_k)$ *has an abstract over-approximated counterpart* $\boldsymbol{expr}_k^\sharp(C_1^\sharp, \ldots, C_k^\sharp)$:
$$\gamma(\boldsymbol{expr}_k^\sharp(C_1^\sharp, \ldots, C_k^\sharp)) \supseteq \{\ \boldsymbol{expr}_k(c_1, \ldots, c_k) \mid c_i \in \gamma(C_i^\sharp)\ \} \ .$$
7. *If* $\mathcal{C}$ *has strictly infinite chains, there is a* widening *operator* $\triangledown$. $\Diamond$

By strictness, $\gamma(\bot) = \emptyset$. Thanks to points 2 and 3, there exists a unique abstract element $\top$ such that $\gamma(\top) = \mathbb{I}$. The least upper bound $\sqcup$ is also an over-approximation for the union: $\gamma(C_1^\sharp \sqcup C_2^\sharp) \supseteq \gamma(C_1^\sharp) \cup \gamma(C_2^\sharp)$.

3.3 A Few Classical Bases

We now present a few set of bases that allow us to retrieve the non-relational constant propagation [6], interval [5], and congruence domains [14].

Constant Basis. $\mathcal{C}_{\text{cst}} \stackrel{\text{def}}{=} \{\bot,\ \top\} \cup \{\ c^\sharp \mid c \in \mathbb{I}\ \}$.

All abstract operators are straightforward and not discussed here (see [6] for more details). There is no need for a widening operator.

Interval Basis. $\mathcal{C}_{[a,b]} \stackrel{\text{def}}{=} \{\bot\} \cup \{\ [a,b] \mid a \in \mathbb{I} \cup \{-\infty\},\ b \in \mathbb{I} \cup \{+\infty\},\ a \leq b^1\ \}$.

Most abstract operators are straightforward (see [5] for more details). We will only recall here the classical widening operator:

$$[a_1, b_1] \triangledown [a_2, b_2] \stackrel{\text{def}}{=} \left[\begin{cases} a_1 & \text{if } a_1 \leq a_2 \\ -\infty & \text{elsewhere} \end{cases} , \quad \begin{cases} b_1 & \text{if } b_1 \geq b_2 \\ +\infty & \text{elsewhere} \end{cases} \right] .$$

In $\mathbb{Q}$ or $\mathbb{R}$, one can alternatively define the *open interval lattice* $\mathcal{C}_{]a,b[}$ the same way. One can even combine these informations to obtain a basis $\mathcal{C}_{[\![a,b]\!]}$ where each bound may or may not be included.

Congruence Basis. $\mathcal{C}_{a\mathbb{Z}+b}^{\mathbb{Z}} \stackrel{\text{def}}{=} \{\bot\} \cup \{\ (a\mathbb{Z}+b) \mid a \in \mathbb{N}^* \cup \{\infty\},\ b \in \mathbb{Z}\ \}$.

A basis is built on $\mathcal{C}_{a\mathbb{Z}+b}^{\mathbb{Z}}$ thanks to the operators described in Figure 4 (using the definitions of Figure 3). However, for the sake of conciseness, Figure 4 does not present abstract k-ary arithmetic expressions, but the binary plus, which is denoted by the infix $\boxplus$ operator (see [14] for more details). There is no strictly increasing infinite chain, so there is no need for a widening operator.

One may also consider to adapt the definitions of Figures 3 and 4 to *rational congruences* [16]: $\mathcal{C}_{a\mathbb{Z}+b}^{\mathbb{Q}} = \{\bot\} \cup \{\ (a\mathbb{Z}+b) \mid a \in \mathbb{Q}^{>0} \cup \{\infty\},\ b \in \mathbb{Q}\ \}$.

[1] Bounds are part of the interval only if finite. Do not be confused by closed interval notations such as $[a, +\infty]$; the interval cannot contain infinite elements.

In the following, $x, x' \in \mathbb{Z}$ and $y, y' \in \mathbb{N}^* \cup \{\infty\}$:

- $y/y' \overset{\text{def}}{\iff} y$ is a divisor of y' ($\exists k \in \mathbb{N}^*$ such that $y' = ky$), or $y' = \infty$;
- $x \equiv x'\ [y] \overset{\text{def}}{\iff} x \not\equiv x'$ and $y/|x - x'|$, or $x = x'$;
- $\vee$ is the least common multiple, extended by $y \vee \infty \overset{\text{def}}{=} \infty \vee y \overset{\text{def}}{=} \infty$;
- $\wedge$ is the greatest common divisor, extended by $y \wedge \infty \overset{\text{def}}{=} \infty \wedge y \overset{\text{def}}{=} y$.

Fig. 3. Classical arithmetic operators extended to $\mathbb{N}^* \cup \{\infty\}$.

- *Concretization:*
$$\gamma(C) \overset{\text{def}}{=} \begin{cases} \{\ ak + b \mid k \in \mathbb{Z}\ \} & \text{if } C = (a\mathbb{Z} + b),\ a \neq \infty; \\ \{\ b\ \} & \text{if } C = (\infty\mathbb{Z} + b); \\ \emptyset & \text{if } C = \bot. \end{cases}$$
- *Order:*
  - $(a\mathbb{Z} + b) \sqsubseteq (a'\mathbb{Z} + b') \overset{\text{def}}{\iff} a'/a$ and $b \equiv b'\ [a']$.
  - $\bot \sqsubseteq C,\ \forall C \in \mathcal{C}$.
- *Intersection (exact abstract counterpart for the intersection $\cap$):*
  - $$(a\mathbb{Z} + b) \sqcap (a'\mathbb{Z} + b') \overset{\text{def}}{=} \begin{cases} (a \vee a')\ \mathbb{Z} + b'' & \text{if } b \equiv b'\ [a \wedge a'], \\ \bot & \text{elsewhere,} \end{cases}$$
    where b'' is such that $b'' \equiv b\ [a \vee a'] \equiv b'\ [a \vee a']$ (Bezout Theorem).
  - $\bot \sqcap C \overset{\text{def}}{=} C \sqcap \bot \overset{\text{def}}{=} \bot,\ \forall C \in \mathcal{C}$.
- *Least Upper Bound:*
  - $(a\mathbb{Z} + b) \sqcup (a'\mathbb{Z} + b') \overset{\text{def}}{=} (a \wedge a' \wedge |b - b'|)\ \mathbb{Z} + \min(b, b')$.
  - $\bot \sqcup C \overset{\text{def}}{=} C \sqcup \bot \overset{\text{def}}{=} C,\ \forall C \in \mathcal{C}$.
- *Sum (exact abstract counterpart for the binary + operator):*
  - $(a\mathbb{Z} + b) \boxplus (a'\mathbb{Z} + b') \overset{\text{def}}{=} (a \wedge a')\ \mathbb{Z} + (b + b')$.
  - $\bot \boxplus C \overset{\text{def}}{=} C \boxplus \bot \overset{\text{def}}{=} \bot,\ \forall C \in \mathcal{C}$.

Fig. 4. Concretization and abstract operators in $\mathcal{C}^{\mathbb{Z}}_{a\mathbb{Z}+b}$.

3.4 Building Non-relational Domains from Bases

Building a non-relational domain $(\mathcal{D}, \preceq)$ from a basis $(\mathcal{C}, \sqsubseteq)$ is straightforward:

- We set $\mathcal{D} \overset{\text{def}}{=} \mathcal{V}\ ! \to \mathcal{C}$
- The concretization Γ, order $\preceq$, union $\sqcup$, and widening ∇ are simply point-wise versions of the corresponding operators on the basis.
- Assignments are defined using the abstract counterpart of expressions:
$$(C_1^\sharp, \ldots, C_i^\sharp, \ldots)_{(v_i \leftarrow \mathit{expr})} \overset{\text{def}}{=} (C_1^\sharp, \ldots, \mathit{expr}_N^\sharp(C_1^\sharp, \ldots, C_{N-1}^\sharp), \ldots)\ .$$
- Only non-relational guards ($v_i \in C\ ?$) do some filter job:
$$(C_1^\sharp, \ldots, C_i^\sharp, \ldots)_{(v_i \in C\ ?)} \overset{\text{def}}{=} (C_1^\sharp, \ldots, C_i^\sharp \sqcap C^\sharp, \ldots) \text{ where } \gamma(C^\sharp) \supseteq C\ .$$
In other guard cases, it is safe to use the identity function.

From an implementation point of view, the non-relational domain is simply a generic functor module, and each basis implementation is a module.

4 Building Weakly Relational Domains from Bases

Now we would like to represent relations of the form $v_j - v_i \in \gamma(C)$ where C lives in a basis $\mathcal{C}$ (instead of $v_i \in \gamma(C)$). A plain basis is not sufficient, we will need a way—a so-called *closure*—to propagate relational information. The main result of this paper can be schemed as follows:

basis (with extra hypotheses)	+ closure	$\Longrightarrow$	weakly relational domain

4.1 Hypotheses on the Basis

Not all bases $\mathcal{C}$ are acceptable. We need the following extra hypotheses:

Hypotheses 1. Acceptable Bases.

1. *There exists* exact *abstract counterparts for the intersection* $\sqcap$ *(which should also be a* lower bound *for* $\sqsubseteq$*), unary minus* $\boxminus$*, and binary plus* $\boxplus$ *operators:*
   - $\gamma(x \boxplus y) = \{\ a + b \mid a \in \gamma(x),\ b \in \gamma(y)\ \}$; *(Abstract plus.)*
   - $\gamma(\boxminus x) = \{\ -a \mid a \in \gamma(x)\ \}$; *(Abstract opposite.)*
   - $x \sqcap y \sqsubseteq x, y$, *so* $\gamma(x \sqcap y) = \gamma(x) \cap \gamma(y)$. *(Abstract intersection.)*
2. *Each singleton* $\{c\}$, $c \in \mathbb{I}$ *must be* exactly *represented by an abstract element* $c^\sharp \in \mathcal{C}$*:* $\gamma(c^\sharp) = \{c\}$.
3. *For each finite family* $(x_i)_{i \in I}$,
$$\boxminus_{i \in I} x_i = \bot \Longrightarrow \exists i, j \in I,\ x_i \sqcap x_j = \bot\ .$$
4. $\sqcap$ *distributes* $\boxplus$*: for each family* $(x_i)_{i \in I}$ *and element* x *of* $\mathcal{C}$,
$$\text{if } \boxminus_{i \in I} x_i \ / \neq,\ \text{then } \boxplus_{i \in I} (x \boxplus x_i) = x \boxplus \left(\boxminus_{i \in I} x_i \right)\ .$$
5. $\boxminus$ *distributes* $\boxplus$ *and* $\sqcap$. $\boxplus$ *and* $\sqcap$ *are commutative and associative.* ✠

These hypotheses were stated in order to prove our main theorem, which is the correctness of the closure operator. Remark that Hypotheses 1.3-4 are very strong. The full basis $\mathcal{C} = \mathcal{P}(\mathbb{I})$, for instance, does not respect them.

Remark. There exists resemblance between bases respecting Hypotheses 1 and the graph-theory classical notion of *complete dïoid* [13]—an extension of *exotic algebras*. A complete dïoid is a complete semi-lattice with an addition (our $\sqcap$), and a multiplication (our $\boxplus$) that distributes over the addition. However, full distributivity in dïoids implies that $\bot \boxplus \top = \top$ where we would have preferred $\bot \boxplus \top = \bot$. Thus, in our framework, distributivity is restricted (Hypothesis 1.4).

4.2 Representing Relations

A set of constraints of the form $v_j - v_i \in \gamma(C),\ C \in \mathcal{C}$ is now represented by a *coherent constraint matrix*:

Definition 3. Constraint Matrices.

1. *A* constraint matrix $\mathbf{m}$ *is a* $N \times N$ *matrix with elements in* $\mathcal{C}$*; the element* $\mathbf{m}_{ij}$ *represents the constraint* $v_j - v_i \in \gamma(\mathbf{m}_{ij})$.
2. *We suppose, as an implicit constraint, that* $v_0 = 0$*, so that unary constraints* $v_i \in \gamma(\mathbf{m}_{0i})$ *can be represented as* $v_i - v_0 \in \gamma(\mathbf{m}_{0i})$.
3. $\mathbf{m}$ *is* coherent *if* $\forall i, j,\ \mathbf{m}_{ij} = \boxminus \mathbf{m}_{ji}$ *and* $\forall i,\ \mathbf{m}_{ii} = \{0\}$.
4. $\mathbf{m}$ *represents the set (so-called* concretization *of* $\mathbf{m}$*):*
   $\Gamma(\mathbf{m}) \stackrel{\text{def}}{=} \{\ (x_0, \ldots, x_{N-1}) \in \mathbb{I}^N \mid x_0 = 0,\ \forall i, j,\ x_j - x_i \in \gamma(\mathbf{m}_{ij})\ \}\ .$ ◊

Our abstract domain is the set $\mathcal{D}$ of coherent constraint matrices, ordered by the point-wise extension $\preceq$ of the partial order $\sqsubseteq$ on $\mathcal{C}$:

$$\begin{aligned} \mathbf{m} \preceq \mathbf{n} &\stackrel{\text{def}}{\Longleftrightarrow} \forall i, j,\ \mathbf{m}_{ij} \sqsubseteq \mathbf{n}_{ij}; \\ \mathbf{m} \doteq \mathbf{n} &\stackrel{\text{def}}{\Longleftrightarrow} \forall i, j,\ \mathbf{m}_{ij} = \mathbf{n}_{ij}; \\ \dot{\bot} &\stackrel{\text{def}}{=} \inf_{\preceq} \mathcal{D} \text{ is such that } \forall i, j,\ \dot{\bot}_{ij} = \bot\ . \end{aligned}$$

The concretization function on $\mathcal{D}$ is Γ, and we have:

Theorem 4. Monotony of Γ.

1. $\mathbf{m} \preceq \mathbf{n} \Longrightarrow \Gamma(\mathbf{m}) \subseteq \Gamma(\mathbf{n})$.
2. $\mathbf{m} \doteq \mathbf{n} \Longrightarrow \Gamma(\mathbf{m}) = \Gamma(\mathbf{n})$. □

However, this is *not an equivalence* and we can have two different constraint matrices $\mathbf{m} \neq \mathbf{n}$ with the same concretization $\Gamma(\mathbf{m}) = \Gamma(\mathbf{n})$.

4.3 General Closure Operator

Implicit Constraints. Because our abstract domain is relational, the constraints between variables are not independent. One can deduce a constraint on $x - z$ by adding a constraint on $x - y$ to a constraint on $y - z$. Such deduced constraints are called *implicit constraints* because they may not be present explicitly in $\mathbf{m}$. More generally, given any path $\langle i = i_1, \ldots, i_n = j \rangle$ in $\mathbf{m}$, we can construct the following implicit constraint:

$$v_j - v_i \in \gamma\left(\boxplus_{l=1}^{n-1} \mathbf{m}_{i_l\, i_{l+1}}\right)\ .$$

Shortest-Path Closure. A nice property of DBMs [19] and periodicity graphs [27] that will hold for our constraint matrices is that the concretization is entirely determined by the set of implicit constraints of the above form. DBMs use any *shortest-path closure* algorithm in order to make all implicit constraints explicit. Here, we adapt the *Floyd-Warshall algorithm* [4, §25.3], to our matrices.

Definition 5. Closure. *Let* $\mathbf{m}$ *be a coherent matrix. Its* closure *is the result* $\mathbf{m}^\bigstar$ *of the following* modified Floyd-Warshall algorithm*:*

$$\begin{cases} \mathbf{m}^0 & \stackrel{\text{def}}{=} \mathbf{m}; \\ \mathbf{m}^{k+1}_{ij} & \stackrel{\text{def}}{=} \mathbf{m}^k_{ij} \sqcap (\mathbf{m}^k_{ik} \boxplus \mathbf{m}^k_{kj}); \\ \mathbf{m}^\bigstar & \stackrel{\text{def}}{=} \mathbf{m}^N \ . \end{cases}$$

$\Diamond$

The Floyd-Warshall algorithm was chosen because it is easy to understand, straightforward to implement, and easy to adapt to constraint matrices. It performs $\mathcal{O}(N^3)$ elementary basis operations.

Here is the main theorem of this paper. The following results will be used extensively in Section 4.4 in order to design abstract operators. The proof of this theorem relies heavily on Hypotheses 1—in fact, the proof itself motivated the hypotheses.

Theorem 6. Closure.

1. $\Gamma(\mathbf{m}^\bigstar) = \Gamma(\mathbf{m})$.
2. $\Gamma(\mathbf{m}) = \emptyset \iff \exists i,\ \mathbf{m}^\bigstar_{ii} = \bot$.
3. *If* $\Gamma(\mathbf{m}) \neq \emptyset$, *then* $\mathbf{m}^\bigstar$ *enjoys the following properties:*
   - $\mathbf{m}^\bigstar$ *is a coherent matrix;* *(Coherence.)*
   - $\forall i, j,\ \mathbf{m}^\bigstar_{ij} = \bigsqcap_{\langle i=i_1,\dots,i_n=j\rangle} \boxplus_{l=1}^{n-1} \mathbf{m}_{i_l\, i_{l+1}}$; *(Transitive closure.)*
   - $\forall i, j,\ \forall c \in \gamma(\mathbf{m}^\bigstar_{ij}),\ \exists (x_0, \dots, x_{N-1}) \in \Gamma(\mathbf{m}),\ x_j - x_i = c$; *(Saturation.)*
   - $\mathbf{m}^\bigstar = \inf_{\preceq}\{\ \mathbf{n} \mid \Gamma(\mathbf{m}) = \Gamma(\mathbf{n})\ \}$; *(Normal form.)*
   - $\mathbf{m}^{\bigstar\bigstar} = \mathbf{m}^\bigstar$. *(Closure.)*

$\square$

Incremental Closure. When modifying slightly a closed matrix, we do not need to perform the modified Floyd-Warshall algorithm completely to get the closure of the new matrix. If the upper-left $M \times M$ sub-matrix of $\mathbf{m}$ is already closed, we can use the following $\mathcal{O}((N-M) \cdot N^2)$ algorithm:

$$\begin{cases} \mathbf{m}^0 & \stackrel{\text{def}}{=} \mathbf{m}; & \\ \mathbf{m}^{k+1}_{ij} & \stackrel{\text{def}}{=} \mathbf{m}^k_{ij} & \text{if } i, j, k < M; \\ \mathbf{m}^{k+1}_{ij} & \stackrel{\text{def}}{=} \mathbf{m}^k_{ij} \sqcap (\mathbf{m}^k_{ik} \boxplus \mathbf{m}^k_{kj}) & \text{elsewhere;} \\ \mathbf{m}^\bigstar & \stackrel{\text{def}}{=} \mathbf{m}^N \ . & \end{cases}$$

We can adapt easily the algorithm—permuting variables—to get a general incremental closure algorithm performing $\mathcal{O}(N^2 \cdot c)$ elementary basis operations, where c is the number of lines and columns that have changed since the last closure.

4.4 Generic Operators

Emptiness Testing. Testing the satisfiability of a constraint matrix is done using Theorem 6.2. Unlike the constraint programming approach, we do not use a specific loop-based satisfiability algorithm, but let our generic closure algorithm solve both the satisfiability and the normal form problems at once.

Equality, Inclusion Testing. The normal form property of Theorem 6.3 allows us to easily test equality and inclusion of non-empty concretizations:

Theorem 7. Equality and Inclusion Testing.
1. $\Gamma(\mathbf{m}) = \Gamma(\mathbf{n}) \iff \mathbf{m}^\bigstar \doteq \mathbf{n}^\bigstar$.
2. $\Gamma(\mathbf{m}) \subseteq \Gamma(\mathbf{n}) \iff \mathbf{m}^\bigstar \preceq \mathbf{n}$. □

Remark that we do not need to close the right argument while testing inclusion.

Union, Intersection. $\gamma(\mathcal{C})$ is stable under intersection, so we simply extend point-wisely $\sqcap$ to represent the intersection of two concretizations:

$$[\mathbf{m} \dot{\sqcap} \mathbf{n}]_{ij} \stackrel{\text{def}}{=} \mathbf{m}_{ij} \sqcap \mathbf{n}_{ij} \ .$$

Theorem 8. Intersection. $\Gamma(\mathbf{m} \dot{\sqcap} \mathbf{n}) = \Gamma(\mathbf{m}) \cap \Gamma(\mathbf{n})$. □

$\gamma(\mathcal{C})$ is not generally closed under union, neither is $\Gamma(\mathcal{D})$. However, if there exists an upper bound $\sqcup$ in $\mathcal{C}$, we can extend it point-wisely in $\mathcal{D}$:

$$[\mathbf{m} \dot{\sqcup} \mathbf{n}]_{ij} \stackrel{\text{def}}{=} \mathbf{m}_{ij} \sqcup \mathbf{n}_{ij} \ .$$

If $\sqcup$ is a *least* upper bound, $\dot{\sqcup}$ can be used to determine the least upper bound of two concretizations, *provided the arguments are closed matrices.*

Theorem 9. (Least) Upper Bound.
1. *If* $\forall a, b \in \mathcal{C},\ \gamma(a \sqcup b) \supseteq \gamma(a) \cup \gamma(b)$, *then* $\Gamma(\mathbf{m} \dot{\sqcup} \mathbf{n}) \supseteq \Gamma(\mathbf{m}) \cup \Gamma(\mathbf{n})$. *(Upper bound.)*
2. *If* $\gamma(a \sqcup b) = \inf_{\subseteq}\{\ \gamma(c) \mid \gamma(c) \supseteq \gamma(a) \cup \gamma(b)\ \}$, *then* $\Gamma(\mathbf{m}^\bigstar \dot{\sqcup} \mathbf{n}^\bigstar) = \inf_{\subseteq}\{\ \Gamma(\mathbf{o}) \mid \Gamma(\mathbf{o}) \supseteq \Gamma(\mathbf{m}) \cup \Gamma(\mathbf{n})\ \}$. *(Least upper bound.)*
3. $(\mathbf{m}^\bigstar \dot{\sqcup} \mathbf{n}^\bigstar)^\bigstar = \mathbf{m}^\bigstar \dot{\sqcup} \mathbf{n}^\bigstar$. *($\dot{\sqcup}$ respects closure.)*

□

Widening. $\mathcal{D}$ has infinite strictly increasing chains only if $\mathcal{C}$ has. A widening $\dot{\triangledown}$ is obtained on $\mathcal{D}$ by point-wise application of the widening $\triangledown$ on $\mathcal{C}$:

$$[\mathbf{m} \dot{\triangledown} \mathbf{n}]_{ij} \stackrel{\text{def}}{=} \mathbf{m}_{ij} \triangledown \mathbf{n}_{ij} \ .$$

$\dot{\triangledown}$ respects Definition 1. Thus, the least fixpoint of an operator F can be over-approximated by the limit of the stationary sequence $X_{i+1} = X_i \dot{\triangledown} F(X_i)$. One could expect, as for the least upper bound, to get a better precision by closing the arguments of $\dot{\triangledown}$, but this is not the case. Even worse, enforcing the closure of the chain by computing $X_{i+1} = (X_i \dot{\triangledown} F(X_i))^\bigstar$ *breaks the ascending chain condition* and prevents the analysis from terminating in some cases. We advocate here the use of the following iteration: $X_{i+1} = X_i \dot{\triangledown} F(X_i^\bigstar)$.

Guard. We can easily implement tests of the form $(v_j - v_i \in C\ ?)$:

$$\left[\mathbf{m}_{(v_j - v_i \in C\ ?)}\right]_{kl} \stackrel{\text{def}}{=} \begin{cases} \mathbf{m}_{kl} \sqcap C^\sharp & \text{if } (k,l) = (i,j); \\ \mathbf{m}_{kl} \sqcap (\boxminus C^\sharp) & \text{if } (k,l) = (j,i); \\ \mathbf{m}_{kl} & \text{elsewhere;} \end{cases}$$
$$\text{choosing } C^\sharp \text{ such that } \gamma(C^\sharp) \supseteq C\ .$$

Tests of the form $(v_j \in C\ ?)$ are implemented by choosing $i = 0$.
For other tests, it is safe to do nothing:

$$\mathbf{m}_{(?)} \stackrel{\text{def}}{=} \mathbf{m}\ .$$

Projection. In order to find the set of values that a variable can take, we use the following theorem derived from the saturation property of the closure:

Theorem 10. $\{\ x \mid \exists (x_0, \dots, x_{N-1}) \in \mathit{\Gamma}(\mathbf{m})\ \textit{with}\ x_i = x\ \} = \gamma(\mathbf{m}^\bigstar_{0i})\ .$ □

Forget. Forgetting the value of a variable is useful to implement the random assignment $(v_i \leftarrow ?)$, which also serves as a coarse approximation for complex assignments. Before forgetting all information on a variable, one should close the argument matrix so that we do not loose implicit constraints:

$$\left[\mathbf{m}_{(v_i \leftarrow ?)}\right]_{kl} \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } k = i \text{ or } l = i; \\ \mathbf{m}^\bigstar_{kl} & \text{elsewhere}\ . \end{cases}$$

Theorem 11.
$\mathit{\Gamma}(\mathbf{m}_{(v_i \leftarrow ?)}) = \{\ (x_0, \dots, x_i, \dots) \mid \exists x,\ (x_0, \dots, x, \dots) \in \mathit{\Gamma}(\mathbf{m})\ \}\ .$ □

Assignment. For assignments of the form $(v_i \leftarrow v_j + c)$, one can find an *exact* abstract counterpart:

$$\left[\mathbf{m}_{(v_i \leftarrow v_i + c)})\right]_{kl} \stackrel{\text{def}}{=} \begin{cases} \mathbf{m}_{kl} \boxplus \{c\} & \text{if } k = i \text{ and } l \neq i; \\ \mathbf{m}_{kl} \boxplus \{-c\} & \text{if } l = i \text{ and } k \neq i; \\ \mathbf{m}_{kl} & \text{elsewhere;} \end{cases}$$

$$\mathbf{m}_{(v_i \leftarrow v_j + c)} \stackrel{\text{def}}{=} (\mathbf{m}_{(v_i \leftarrow ?)})_{(v_i - v_j \in \{c\}\ ?)} \qquad \text{when } i \neq j\ .$$

For generic assignments $(v_i \leftarrow \mathtt{expr}(v_1, \dots, v_{N-1}))$, one can always fall back to imprecise non-relational analysis, first projecting the variables, then using the abstraction $\mathtt{expr}^\sharp$ of $\mathtt{expr}$ in our basis:

$$\mathbf{m}_{(v_i \leftarrow \mathtt{expr}(v_1, \dots, v_{N-1}))} \stackrel{\text{def}}{=} (\mathbf{m}_{(v_i \leftarrow ?)})_{(v_i - v_j \in \gamma(C^\sharp)\ ?)}$$
$$\text{where } C^\sharp \stackrel{\text{def}}{=} \mathtt{expr}^\sharp(\mathbf{m}^\bigstar_{01}, \dots, \mathbf{m}^\bigstar_{0(N-1)})\ .$$

Trying to be the most precise in all cases may lead to complex algorithms. It seems only worth trying to be a little more precise in some widespread cases, such as $(v_i \leftarrow v_j + v_k)$, for instance:

$$\mathbf{m}_{(v_i \leftarrow v_j + v_k)} \stackrel{\text{def}}{=} (\mathbf{m}_{(v_i \leftarrow\ ?)})_{(v_i \in \gamma(\mathbf{m}^\bigstar_{0j} \boxplus \mathbf{m}^\bigstar_{0k})\ ?)\ (v_i - v_j \in \gamma(\mathbf{m}^\bigstar_{0k})\ ?)\ (v_i - v_k \in \gamma(\mathbf{m}^\bigstar_{0j})\ ?)}\ .$$

Interaction with the Closure. Some of the above operators require the matrix argument(s) to be closed. Some do respect closure—the result is closed if the argument(s) is(are)—and some do not (intersection, guard, assignment, etc.). We thus advocate the use of a *lazy* method that remembers when a matrix is in closed form, and recomputes the closure only when needed. When only a few lines and columns of the matrix are changed (guard, assignment, etc.), we can use the incremental closure. It is useless when all coefficients are changed at once (intersection, widening).

4.5 Some Constructed Domains

We are now ready to apply our construction to the bases presented in Section 3.3, thanks to the following theorem:

Theorem 12. *$\mathcal{C}_{\mathrm{cst}}$, $\mathcal{C}_{[a,b]}$, $\mathcal{C}_{]a,b[}$, $\mathcal{C}^{\mathbb{Z}}_{a\mathbb{Z}+b}$, and $\mathcal{C}^{\mathbb{Q}}_{a\mathbb{Z}+b}$ respect Hypotheses 1.* □

Translated Equality Domain. The simplest domain is obtained from the constant basis $\mathcal{C}_{\mathrm{cst}}$ and represents constraints of the form $(v_i = v_j + c)$. This domain is not of great practical interest: its expressive power is low as it is a particular case of the following two domains. It is possible that more efficient solutions exist, as we are not very far from simple equality constraints $v_i = v_j$ for which very efficient algorithms are known (such as, the *Union-Find* algorithm [4, §22]).

Zone Domain. In order to represent invariants of the form $(v_i - v_j \leq c)$, one can think of the basis of initial segments $\{\,]-\infty, a] \mid a \in \mathbb{I} \cup \{+\infty\}\}$, but initial segments are not closed under the $\boxminus$ operation (Hypothesis 1.1). Completing this basis, one naturally find the interval basis $\mathcal{C}_{[a,b]}$.

Compared to classical DBMs [22], the domain obtained is a little redundant (each constraint is represented twice), but has exactly the same expressiveness and complexity. It has the advantage of being implemented over any existing interval library, greatly reducing the need for programming. One can also enhance the zone domain in $\mathbb{Q}$ and $\mathbb{R}$ using the $\mathcal{C}_{]a,b[}$ basis that manipulates both strict and non strict constraints.

Zone-Congruence Domain. Using the integer congruence basis $\mathcal{C}^{\mathbb{Z}}_{a\mathbb{Z}+b}$, one builds a domain that discovers constraints of the form $(v_i - v_j \equiv a\ [b])$. This construction looks like *periodicity graphs* [27], but we treat here the case of least upper bound and general purpose transfer functions in detail, whereas [27] is only interested in satisfiability, normal form and conjunction. Moreover, we feel that [27] misses the correct proof of the normal form theorem (our Theorem 6) and does not understand that it relies on some strong properties of congruence sets (Hypotheses 1). Our framework can also extend this domain to a domain of *rational congruences*: $(v_i - v_j \equiv a\ [b])$ with $a, b \in \mathbb{Q}$.

Product Domain. *Reduced product* is a well-known technique [6] for improving the precision of an analysis by combining the power of two abstract domains. It often gives better results than two separate analyses, because it conveys information from one domain to the other during the analysis via a so-called *reduction procedure*, which is a couple of binary operators $(\circlearrowleft, \circlearrowright)$ such that:

$$\begin{array}{l} \circlearrowleft\colon \mathcal{D}_1 \times \mathcal{D}_2 \mapsto \mathcal{D}_1, \\ \circlearrowright\colon \mathcal{D}_1 \times \mathcal{D}_2 \mapsto \mathcal{D}_2, \end{array} \quad \begin{cases} C_1 \circlearrowleft C_2 \preceq_1 C_1; \\ C_1 \circlearrowright C_2 \preceq_2 C_2; \\ \Gamma_1(C_1 \circlearrowleft C_2) \cap \Gamma_2(C_1 \circlearrowright C_2) = \Gamma_1(C_1) \cap \Gamma_2(C_2) \ . \end{cases}$$

In our case, the reduction can be defined on bases—as long as Hypotheses 1 are not broken—with the exact same precision benefit. Moreover, reductions are easier to design on non-relational bases. For example, if we use the following reduction between $\mathcal{C}_{[a,b]}$ and $\mathcal{C}^{\mathbb{Z}}_{a\mathbb{Z}+b}$, we obtain a basis allowing the construction of a domain for constraints of the form $(v_i - v_j \in a \cdot [b,c] + d)$:

$$\begin{cases} [a,b] \circlearrowleft (c\mathbb{Z}+b) \stackrel{\text{def}}{=} [\ \min\{x \in (c\mathbb{Z}+d) \mid x \geq a\}, \\ \qquad\qquad\qquad\qquad \max\{x \in (c\mathbb{Z}+d) \mid x \leq b\}\ ]; \\ [a,b] \circlearrowright (c\mathbb{Z}+d) \stackrel{\text{def}}{=} (c\mathbb{Z}+d) \ . \end{cases}$$

Failure. So far, all seems to work well. However, one can find some bases used in very common abstract domains that do not respect Hypotheses 1. For example, the *sign* basis [6] $\mathcal{C}_{\pm} = \{\ \bot,\ ]-\infty,0],\ [0,0],\ [0,+\infty[,\ ]-\infty,+\infty[\ \}$ and the open interval basis $\mathcal{C}_{]a,b[}$ do not respect Hypothesis 1.2. The *interval congruence* basis $\mathcal{C}_{a\mathbb{Z}+[b,c]}$ (introduced in [20]) does not respect Hypothesis 1.1 (it is not stable under intersection). We do not know if it is possible to build weakly relational domains from such bases.

Modularity. As for non-relational domains, the weakly relational domain family is simply a generic module functor, taking the very same bases implementation modules as parameter.

5 Applications and Future Work

Applications. So far, this framework has only been implemented as an OCaml prototype and tried on a few toy examples. At program point (•) of the program in Figure 1, the reduced-product of the zone and zone-congruence domains found the invariant $(x \leq 5,\ x \equiv 1\ [2])$, which is almost as good as polyhedron and congruence equality analyses combined (Figure 2). It failed to discover that $(x \geq -5)$; however, the octagon abstract domain of [23] that also uses graph-based algorithms can do it.

If in the program of Figure 1 the constant 5 is replaced by a variable m the value of which is not known at analysis time, the analyzer still finds the precise symbolic invariant $(x \leq m,\ x \equiv m\ [2])$.

Scalability. It is still unknown whether graph-based abstract domains scale up. Because of the quadratic memory cost, it cannot handle all the variables of a large program at once; one has to split this set into *packets* in which relational information might be important. These packets do not need to be disjoint, and one can use pivot variables to transfer information between packets. We are currently investigating on such methods.

Because our domain family is relational, it is also adapted to symbolic and modular analyses. One can cut down the cost of an analysis and make it incremental by analyzing separately small pieces of a program [8].

Theoretical Extensions. We tried, in this article, to unite some graph-based numerical satisfiability algorithm and extend them up to an abstract domain, in a united framework. However, a few graph-based algorithms are not handled here: the octagon abstract domain [23] ($\pm x \pm y \leq c$ constraints) and Shostak's satisfiability algorithms [26] ($\alpha x + \beta y \leq c$). It would be interesting to unite all those in a general framework and derive a numerical abstract domain for constraints of the form ($\alpha x + \beta y \leq c$).

6 Conclusion

In this paper, we have proposed the systematic construction of a family of relational domains that represent and manipulate constraints of the form ($x - y \in C$). This construction can be seen as a *functor* lifting non-relational domains to relational ones. The memory cost of an abstract state is *quadratic*, and each transfer function application performs, at worse, a *cubic* number of operations in the non-relational domain. The crux of the method is the adaptation of the shortest-path closure algorithm to a normal form, allowing the derivation of most abstract operators and transfer functions.

In this framework, we have successfully retrieved the existing DBM domain, and constructed new ones. It is the author's opinion that these domains fill a precision and complexity gap between former non-relational and relational domains, and can be used to design medium cost, yet precise, analyses.

References

[1] F. Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In Springer-Verlag, editor, *PLILP'90*, volume 456 of *LNCS*, pages 307–323, 1990.

[2] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *FMPA'93*, number 735 in LNCS, 1993.

[3] M. Codish and C. Taboch. A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In *ALP'98*, volume 1298 of *LNCS*, pages 31–45. Springer-Verlag, September 1997.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[5] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *ISOP'76*, pages 106–130. Dunod, Paris, France, 1976.
[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM POPL'77*, pages 238–252. ACM Press, 1977.
[7] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.
[8] P. Cousot and R. Cousot. Modular static program analysis, invited paper. In *CC'02*, number 2304 in LNCS, pages 159–178, 2002.
[9] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM POPL'78*, pages 84–97. ACM Press, 1978.
[10] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *ACM PLDI'94*, pages 230–241. ACM Press, 1994.
[11] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *SAS'01*, number 2126 in LNCS, July 2001.
[12] J. Feret. Occurrence counting analysis for the π-calculus. In *GETCO'00*, volume 39.2 of *BRICS NS-00-3*, 2001.
[13] M. Gondran and M. Minoux. *Graphs and Algorithms*. Wiley, 1984.
[14] P. Granger. Static analysis of arithmetical congruences. In *International Journal of Computer Mathematics*, volume 30, pages 165–190, 1989.
[15] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT'91*, number 493 in LNCS, pages 169–192, 1991.
[16] P. Granger. Static analyses of congruence properties on rational numbers. In *SAS'97*, volume 1302 of *LNCS*, pages 278–292, 1997.
[17] W. Harvey and P. Stuckey. A unit two variable per inequality integer constraint solver for constraint logic programming. In *ACSC'97*, volume 19, pages 102–111, February 1997.
[18] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, pages 133–151, 1976.
[19] K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *IEEE RTSS'97*, pages 14–24. IEEE CS Press, December 1997.
[20] F. Masdupuy. Semantic analysis of interval congruences. In *FMPTA'93*, volume 735 of *LNCS*, pages 142–155, 1993.
[21] L. Mauborgne. *Representation of Sets of Trees for Abstract Interpretation*. PhD thesis, École Polytechnique, Palaiseau, France, November 1999.
[22] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO II*, volume 2053 of *LNCS*, pages 155–172. Springer-Verlag, May 2001.
[23] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.
[24] D. Monniaux. An abstract Monte-Carlo method for the analysis of probabilistic programs. In *POPL'01*, number 1824 in ACM, pages 93–101, 2001.
[25] V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, Cambridge, September 1977.
[26] R. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981.
[27] D. Toman and J. Chomicki. Datalog with integer periodicity constraints. In *Journal of Logic Programming*, pages 189–203. The MIT Press, 1994.

Static Analysis of the Numerical Stability of Loops

Matthieu Martel

CEA - Recherche Technologique
LIST-DTSI-SLA
CEA F91191 Gif-Sur-Yvette Cedex, France
mmartel@cea.fr

Abstract. We introduce a relational static analysis to determine the stability of the numerical errors arising inside a loop in which floating-point computations are carried out. This analysis is based on a stability test for non-linear functions and on a precise semantics for floating-point numbers that computes the propagation of the errors made at each operation. A major advantage of this approach is that higher-order error terms are not neglected. We introduce two algorithms for the analysis. The first one, less complex, only determines the global stability of the loop. The second algorithm determines which particular operation makes a loop unstable. Both algorithms have been implemented and we present some experimental results.

Keywords: Abstract Interpretation, Numerical Precision, Relational Analysis, Semantics of Floating-point Numbers.

1 Introduction

It is often hard for a programmer to understand how precise the result of a floating-point calculation is and to understand which operations introduce the most significant errors [7,11]. Recent work has shown that abstract interpretation [6] is a good candidate for the validation and debugging of numerical codes [9,13] and the first attempts to verify industrial programs with a prototype analyzer are promising [10]. This approach consists in defining a precise concrete semantics for floating-point operations, based on the IEEE 754 Norm [2,7], and, next, in practice, using a more compact abstract semantics, among the many possible ones, to check properties of interest for a program [13].

To our knowledge, most of the alternative techniques aim at dynamically estimating a better approximation of the real numbers that the program would output if the machine were working in infinite precision [12,16]. On the contrary, since we are interested in detecting the errors possibly introduced by floating-point numbers, we always work with the values used by the machine and we

[1] This work was supported by the RTD project IST-1999-20527 "DAEDALUS" of the European FP5 programme.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 133–150, 2002.
© Springer-Verlag Berlin Heidelberg 2002

compute the errors attached to them. In addition, as our approach is based on abstract interpretation, our estimations are valid for a large class of executions.

Concerning the abstract interpretation of numerical programs, the case of calculations carried out in loops, addressed in this article, requires particular attention because of the following reasons. First, loops introduce the most subtle and possibly large imprecision because the errors arising in one iteration are related to the ones of the previous iterations. Small errors arising in one iteration can be accumulated, magnified or made negligible in the following steps. Second, for many stable loops, i.e. loops in which the errors decrease at each iteration, the errors arising in the first iterations may be significant. A static analyzer must unfold such loops a finite but a priori unknown number of times in order to detect their stability. Otherwise, using widening operators, the analyzer might state that arbitrarily large errors can be made at this location and, consequently, might state that the loop is unstable. Third, the usual narrowing techniques cannot be used to improve the results obtained by widening. This is due to the fact that the information used to narrow the result only concerns the floating-point values of the variables and this cannot be used to reduce the error terms attached to them during the first phase.

We introduce a relational static analysis to detect whether the calculations carried out in a loop are stable, i.e. whether the errors due to the floating-point numbers increase or decrease as the loop is unrolled. An important feature of our analysis is that it does not neglect the higher-order error terms (error terms obtained by multiplying previous error terms), unlike most previous work [12,16]. As shown in Section 3, the higher-order error terms may be large even if the first-order ones are negligible. The analysis is based on the semantics of floating-point numbers with errors [13] and on a stability test for non-linear maps obtained from the calculation of the Lyapunov exponents, a widely used technique to determine the stability of dynamic systems [1,14]. Intuitively, this test consists of determining whether the semantic equations describing the errors at each iteration of a loop are contracting or expanding. The semantic equations are automatically generated from the syntax of the program and we introduce two algorithms to determine their stability. The first one involves fewer calculations but only computes the global stability of the loop by checking that no error term diverges. The second algorithm precisely computes the stability of each error term. So, if an instability is detected, the second algorithm indicates which operation in the loop is responsible for the loss of precision. Both algorithms have been implemented and we comment on some experimental results.

The semantics of floating-point numbers with errors is given in Section 2. The Lyapunov test used by the analysis is introduced in Section 3 and we show in Section 4 how to generate the semantic equations needed for the test. Finally, in Section 5 we introduce the two algorithms to determine the stability of a loop and we comment on some results obtained from our implementation.

2 Floating-Point Numbers with Errors

This section briefly introduces the semantics of floating-point numbers with errors which is detailed in ref. [9,13]. As an introductory example, let us ex-

amine a simple calculation involving two values $a_{\mathbb{F}} = 621.3$ and $b_{\mathbb{F}} = 1.287$. For the sake of simplicity, we assume that $a_{\mathbb{F}}$ and $b_{\mathbb{F}}$ belong to a simplified set of floating-point numbers composed of a mantissa of four digits written in base 10. We assume that initial errors are attached to $a_{\mathbb{F}}$ and $b_{\mathbb{F}}$, and we write $a = 621.3\varepsilon + 0.05\varepsilon_1$ and $b = 1.287\varepsilon + 0.0005\varepsilon_2$ to indicate that the value of the initial error on $a_{\mathbb{F}}$ (resp. $b_{\mathbb{F}}$) is 0.05 (resp. 0.0005). ε is a formal variable attached to the floating-point coefficient of a and b and ε_1 and ε_2 are formal variables related to static control points. a and b are called *floating-point numbers with errors*. Let us focus on the product $a_{\mathbb{F}} \times b_{\mathbb{F}}$ whose exact result is $a_{\mathbb{F}} \times b_{\mathbb{F}} = 799.6131$. This calculation carried out with floating-point numbers with errors yields $a \times b = 799.6131\varepsilon\varepsilon + 0.06435\varepsilon\varepsilon_1 + 0.31065\varepsilon\varepsilon_2 + 0.000025\varepsilon_1\varepsilon_2$ which we rewrite as $a \times b = 799.6131\varepsilon + 0.06435\varepsilon_1 + 0.31065\varepsilon_2 + 0.000025\varepsilon_{hi}$. The rewriting step is made to keep only one formal variable per coefficient and obeys the following rule. The indices of the formal variables ε_1, ε_2 etc. are viewed as words of the alphabet of the control points and the product of two variables yields a new formal variable by concatenation of the index words. The empty word is the index of the variable ε related to the floating-point component of the number, a word of length one describes a first-order error and the special word hi is used to identify all the words of length greater than one. We assume $\text{length}(hi) > 1$. The product of formal variables is summed up by the rule:

$$\varepsilon_u \times \varepsilon_v = \begin{cases} \varepsilon_{uv} \text{ if length}(uv) \leq 1 \\ \varepsilon_{hi} \text{ otherwise} \end{cases} \tag{1}$$

The difference between $a_{\mathbb{F}} \times b_{\mathbb{F}}$ and 621.35×1.2875 is 0.375025 and this error stems from the fact that the initial error on a (resp. b) was multiplied by b (resp. a) and that a second-order error corresponding to the multiplication of both errors was introduced. So, at the end of the calculation, the contribution to the global error of the initial error on a (resp. b) is 0.06435 (resp. 0.31065) and corresponds to the coefficient attached to the formal variable ε_1 (resp. ε_2). Finally, the number 799.6131 has too many digits to be representable in our floating-point number system and we refer to the IEEE-754 norm for floating-point arithmetic to determine how the values are rounded [2]. Let $\mathbb{F}$ be either the set of simple or double precision floating-point numbers. The norm fully specifies the function $\uparrow_\circ : \mathbb{R} \to \mathbb{F}$ which returns the roundoff of a real number $r \in \mathbb{R}$ with respect to the current rounding mode $\circ$ [2,7]. In addition, it ensures that the elementary operations are correctly rounded, i.e. for any operator $\diamond \in \{+, -, \times, \div\}$, we have:

$$\forall f_1, f_2 \in \mathbb{F},\ f_1 \diamond_{\mathbb{F}} f_2 = \uparrow_\circ (f_1 \diamond_{\mathbb{R}} f_2) \tag{2}$$

In Equation (2), $\diamond_{\mathbb{F}}$ and $\diamond_{\mathbb{R}}$ denote the same operation over $\mathbb{F}$ and $\mathbb{R}$. For our needs, we also introduce the function $\downarrow_\circ : \mathbb{R} \to \mathbb{R}$ defined by $\downarrow_\circ r = r - \uparrow_\circ r$. Assuming that the machine we are using conforms to that standard, we may claim that the computed floating-point number for our example is $\uparrow_\circ (799.6131) = 799.6$ and that a new first error term $0.0131\varepsilon_\times$ is introduced by the multiplication. To sum up, we have

$$a \times b = 799.6\varepsilon + 0.06435\varepsilon_1 + 0.31065\varepsilon_2 + 0.000025\varepsilon_{12} + 0.0131\varepsilon_\times$$

At first sight, one would think that the precision of the calculation mainly depends on the initial error on a since it is 100 times larger than the one on b. However the above result indicates that the final error is mainly due to the initial error on b. Hence, to improve the precision of the final result, one should first try to increase the precision on b (whenever possible).

We now introduce the formal semantics of floating-point numbers with errors assuming that the control points of the program are annotated by unique labels $\ell \in L$ and that $\mathcal{L}$ denotes the set of words on L of length at most 1 plus the special word hi used to denote the higher-order errors (and which corresponds to no particular control point). This comes from ref. [13] which introduces more general semantics for floating-point numbers with errors. Errors terms of order n correspond to words of length n and the empty word is related to the term for the floating-point number. The multiplication of terms of order m and n yields a new term of order $m+n$ denoted by a word of length $m+n$. In this article, hi identifies all the words of length greater that one and concatenation is defined by Equation (1). Words ℓ, ℓ_0, hi etc. are written as exponents of expressions, e.g. e^{ℓ}. A floating-point number r occurring at the control point ℓ_0 is represented by the series

$$r^{\ell_0} = f\boldsymbol{\varepsilon} + \sum_{\ell \in \mathcal{L}} \omega^{\ell} \boldsymbol{\varepsilon}_{\ell} \tag{3}$$

In Equation (3), f is the floating-point number approximating the value of r. f is always attached to the formal variable $\boldsymbol{\varepsilon}$ whose index is the empty word. A term $\omega^{\ell}\boldsymbol{\varepsilon}_{\ell}$ denotes the contribution to the global error of the first-order error introduced by the operation labeled ℓ during the evaluation of r. $\omega^{\ell} \in \mathbb{R}$ is the scalar value of this error term and $\boldsymbol{\varepsilon}_{\ell}$ is a formal variable.

The elementary operations are defined in Figure 1 for $r_1 = f_1\boldsymbol{\varepsilon}_1 + \sum_{\ell \in \mathcal{L}^+} \omega_1^{\ell}\boldsymbol{\varepsilon}_{\ell}$ and $r_2 = f_2\boldsymbol{\varepsilon}_2 + \sum_{\ell \in \mathcal{L}^+} \omega_2^{\ell}\boldsymbol{\varepsilon}_{\ell}$. $\mathcal{L}^+$ denotes the set $\mathcal{L}$ without the empty word. In addition, the symbols f and ω are used interchangeably to denote the coefficient of the variable $\boldsymbol{\varepsilon}$. The formal series $\sum_{\ell \in \mathcal{L}} \omega^{\ell}\boldsymbol{\varepsilon}_{\ell}$ related to the result of an operation $\diamond^{\ell_i}$ contains the combination of the errors on the operands plus a new error term $\downarrow_{\circ} (f_1 \diamond_{\mathbb{R}} f_2)\boldsymbol{\varepsilon}_{\ell_i}$ corresponding to the error introduced by the operation $\diamond_{\mathbb{F}}$ occurring at point ℓ_i. The rules for addition and subtraction are natural. The elementary errors are added or subtracted componentwise in the formal series and the new error due to point ℓ_i corresponds to the roundoff of the result.

Multiplication requires more care because it introduces higher-order errors due to the multiplication of the first-order errors. For instance, let us consider the multiplication at point ℓ_3 of two initial data $r_1^{\ell_1} = (f_1\boldsymbol{\varepsilon} + \omega_1^{\ell_1}\boldsymbol{\varepsilon}_{\ell_1})$ and $r_2^{\ell_2} = (f_2\boldsymbol{\varepsilon} + \omega_2^{\ell_2}\boldsymbol{\varepsilon}_{\ell_2})$.

$$r_1^{\ell_1} \times^{\ell_3} r_2^{\ell_2} = \uparrow_{\circ} (f_1 f_2)\boldsymbol{\varepsilon} + f_2\omega_1^{\ell_1}\boldsymbol{\varepsilon}_{\ell_1} + f_1\omega_2^{\ell_2}\boldsymbol{\varepsilon}_{\ell_2} + \omega_1^{\ell_1}\omega_2^{\ell_2}\boldsymbol{\varepsilon}_{hi} + \downarrow_{\circ} (f_1 f_2)\boldsymbol{\varepsilon}_{\ell_3} \tag{9}$$

As shown in Equation (9), the floating-point number computed by this multiplication is $\uparrow_{\circ} (f_1 f_2)$. The initial first-order errors $\omega_1^{\ell_1}\boldsymbol{\varepsilon}_{\ell_1}$ and $\omega_2^{\ell_2}\boldsymbol{\varepsilon}_{\ell_2}$ are multiplied by f_2 and f_1 respectively. In addition, the multiplication introduces a new first-order error $\downarrow_{\circ} (f_1 f_2)$ which is attached to the formal variable $\boldsymbol{\varepsilon}_{\ell_3}$ in order to

$$r_1 +^{\ell_i} r_2 \stackrel{\text{def}}{=} \uparrow_\circ (f_1 + f_2)\boldsymbol{\varepsilon} + \sum_{\ell \in \mathcal{L}^+} (\omega_1^\ell + \omega_2^\ell)\boldsymbol{\varepsilon}_\ell + \downarrow_\circ (f_1 + f_2)\boldsymbol{\varepsilon}_{\ell_i} \tag{4}$$

$$r_1 -^{\ell_i} r_2 \stackrel{\text{def}}{=} \uparrow_\circ (f_1 - f_2)\boldsymbol{\varepsilon} + \sum_{\ell \in \mathcal{L}^+} (\omega_1^\ell - \omega_2^\ell)\boldsymbol{\varepsilon}_\ell + \downarrow_\circ (f_1 - f_2)\boldsymbol{\varepsilon}_{\ell_i} \tag{5}$$

$$r_1 \times^{\ell_i} r_2 \stackrel{\text{def}}{=} \uparrow_\circ (f_1 f_2)\boldsymbol{\varepsilon} + \sum_{\substack{\ell_1 \in \mathcal{L} \\ \ell_2 \in \mathcal{L} \\ \ell_1 \cdot \ell_2 \neq \epsilon}} \omega_1^{\ell_1}\omega_2^{\ell_2}\boldsymbol{\varepsilon}_{\ell_1 \cdot \ell_2} + \downarrow_\circ (f_1 f_2)\boldsymbol{\varepsilon}_{\ell_i} \tag{6}$$

$$(r_1)^{-1^{\ell_i}} \stackrel{\text{def}}{=} \uparrow_\circ (f_1^{-1})\boldsymbol{\varepsilon} - \frac{1}{f_1}\sum_{\ell \in \mathcal{L}} \frac{\omega^\ell}{f_1}\boldsymbol{\varepsilon}_\ell + \frac{1}{f_1}\sum_{n \geq 2}(-1)^n \left(\sum_{\ell \in \mathcal{L}} \frac{\omega^\ell}{f_1}\right)^n \boldsymbol{\varepsilon}_{hi} + \downarrow_\circ (f_1^{-1})\boldsymbol{\varepsilon}_{\ell_i} \tag{7}$$

$$r_1 \div^{\ell_i} r_2 \stackrel{\text{def}}{=} r_1 \times^{\ell_i} (r_2)^{-1^{\ell_i}} \tag{8}$$

Fig. 1. Elementary operations for floating-point numbers with errors.

indicate that this error is due to the product occurring at the control point ℓ_3. Finally, this operation also introduces a second-order error that we attach to the formal variable $\boldsymbol{\varepsilon}_{hi}$. In Figure 1, Equation (6) is a generalization of Equation (9).

The term for division is obtained by means of a power series development. The full semantics is defined by the domain of floating-point numbers with errors for values and by the usual operational semantics for the expressions [13].

This semantics details the contribution to the global error of the first-order error terms and globally computes the higher-order error arising during the calculation. In practice, higher-order errors are often negligible. So this semantics allows us to determine the sources of imprecision while checking that the higher-order errors are actually globally negligible. More precise semantics are introduced in ref. [13].

In the rest of this paper, we use an abstract semantics [6], based on the one defined in this section, in which the coefficients f and ω^ℓ are abstracted by intervals of floating-point numbers and intervals of multi-precision floating-point numbers, respectively.

3 Stability in Loops

This section introduces the mathematical tools used to determine the stability of the calculations in loops. Let $f : \mathbb{R}^m \to \mathbb{R}^m$ and let $\boldsymbol{x}_0 \in \mathbb{R}^m$. We want to

study the behavior of the *orbit* of $\boldsymbol{x}_0$ under f, i.e. the sequence $(\boldsymbol{x}_n)_n$ of iterates defined by:

$$\boldsymbol{x}_n = f(\boldsymbol{x}_{n-1}) = f^n(\boldsymbol{x}_0) \tag{10}$$

Our aim is to determine whether the error on $\boldsymbol{x}_0$ as well as the errors arising during the calculation of $(\boldsymbol{x}_n)_n$ increase or decrease at each iteration. For example, let us consider the iterates of the function $g : x \mapsto x^2$ with x_0 in the vicinity of 1. Obviously, the sequence $(g^n(x_0))_n$ converges to 0 if $x_0 < 1$, is constant if $x = 1$ and goes to ∞ if $x > 1$. Moreover, for two distinct initial values $x_0 > 1$ and $x_0' > 1$, $|g^n(x_0) - g^n(x_0')| \to \infty$ as $n \to \infty$. Clearly, in a program computing $(g^n(x_0))_n$, a small error on the initial value x_0, possibly due to the use of floating-point numbers, may widely affect the final result. The map g is said to be *unstable* in the vicinity of 1.

Much work has been done on the theory of iterated function systems (IFS) [1,3,4,14]. For multi-dimensional maps, the space is usually divided in two sets, the *Fatou set* and the *Julia set* respectively containing the points x_0 for which the sequences $(g^n(x_0))_n$ are stable and unstable [4]. For many maps, the limit between both zones is fractal. In this article, we do not directly study the stability of the map f corresponding to the body of a loop. Instead, we study the stability of the semantic equations generated from the loop with respect to the semantics of floating-point numbers with errors introduced in Section 2. In doing so, we not only compute the stability of the floating-point numbers returned by a program but also the stability of the errors made during the floating-point calculations. Let us consider again the function $g : x \mapsto x^2$ and the initial floating-point number with errors $x_0^\ell = 0.95\boldsymbol{\varepsilon} + 0.1\boldsymbol{\varepsilon}_\ell$. With the simple floating-point number $\hat{x}_0 = 0.95$, $(g^n(\hat{x}_0))_n \to 0$ and this sequence is stable in the immediate vicinity of this point. But if no error had been made in the calculation of $\hat{x}_0$, we would have $\hat{x}_0 = 1.05$ and $(g^n(\hat{x}_0))_n \to \infty$. So, the best information an analysis can give to the programmer is that if $x_0^\ell = 0.95\boldsymbol{\varepsilon} + 0.1\boldsymbol{\varepsilon}_\ell$ then the calculation of the iterates of g is stable (since the floating-point number goes to 0 in the vicinity of 0.95) but that the errors made during this calculation are unstable (they are arbitrarily large because of the initial 0.1 error).

The way this function f is generated from the syntax of the program is described in Section 4. For now on, we assume that f is given and we focus on determining its stability. In the particular case where $f : \mathbb{R}^m \to \mathbb{R}^m$ is a linear map (contrary to the example of g), the stability can be deduced from a calculation of its eigenvalues. If we assume that the matrix M of f has distinct real eigenvalues then M is similar to a diagonal matrix:

$$N = \begin{pmatrix} x_1 & & 0 \\ & \ddots & \\ 0 & & x_m \end{pmatrix} \tag{11}$$

and the nth iterate f^n is $N^n = \begin{pmatrix} x_1^n & & 0 \\ & \ddots & \\ 0 & & x_m^n \end{pmatrix}$.Hence, the iterates of f are stable if $0 \leq |x_i| < 1$ for all $1 \leq i \leq m$. This last property still holds if f has repeated or

complex eigenvalues. Here, a static analysis can use standard algorithms to compute an upper approximation of the eigenvalues (see Section 5.1). This approach was suggested by E. Goubault [9].

Unfortunately, the calculations carried out in programs are often non-linear (any multiplication between two non-constant terms is non-linear) and additional non-linearities are introduced in f by the functions modeling the propagation of errors. For instance, the higher-order terms are in general strongly non-linear. In addition, they may be non-negligible, even for simple examples:

$$\hat{g} \ : \ \begin{pmatrix} \hat{x} \\ e_x \\ e_{xh} \end{pmatrix} \mapsto \begin{pmatrix} \hat{x}^2 \\ 2xe_x \\ e_x^2 + e_{xh}^2 + 2\hat{x}e_{xh} + 2e_x e_{xh} \end{pmatrix} \tag{12}$$

$\hat{g}$ is a simplified version of the function generated by our method from a program implementing $g : x \mapsto x^2$. The real number x is divided into three parts, $\hat{x}$ for the floating-point number used instead of x, e_x and e_{xh} for the first-order and higher-order errors made on $\hat{x}$. $\hat{g}$ is obtained from g with $x = \hat{x} + e_x + e_{xh}$. The first dimension is related to the floating-point number, the second and third ones to the first and higher-order errors. Note that, for the sake of simplicity, some error terms are neglected in $\hat{g}$, e.g. the roundoff errors made during the calculations. The full function is introduced later on. Starting with $\hat{x} = 0.95$, $e_x = 0.1$ and $e_{xh} = 0$, one can easily check that the iterates go to $(0, 0, \infty)$. This result means that the floating-point calculations carried out inside the loop tend to 0, that the first-order errors also become null as n increases, but that the higher-order errors become arbitrarily large. Since the higher-order error only is significant, a similar analysis which neglected it would not detect any problem in this calculation.

As a consequence, we keep the non-linear model but there will no longer be a matrix N like the one of Equation (11). The role of the eigenvalues of N is taken by the Lyapunov exponents which measure the rates of separation of the iterates, starting from a point $x_0 \in \mathbb{R}^m$, in m orthogonal directions [1]. The intuition behind this concept is given in the next paragraph.

Let $f : \mathbb{R} \to \mathbb{R}$ and $x_0 \in \mathbb{R}$ such that $f(x_0) = x_0$. x_0 is an *attractive* (resp. a *repelling*) fixed point of f, if the first derivative satisfies $f'(x_0) < 1$ (resp. $f'(x_0) > 1$). If x_0 is repelling, an orbit of f starting from a point in the vicinity of x_0 will separate from x_0 with an approximate rate of $|f'(x_0)|$ by iteration. Otherwise the orbit is attracted by x_0. Now, if x_0 is a periodic point of f, i.e. $x_0 \neq f(x_0) \neq \ldots \neq f^k(x_0) = x_0$, x_0 is an attractive or a repelling periodic point of f depending on the value of $a = |f'(x_0)| \times |f'(x_1)| \times \ldots \times |f'(x_k)|$ with respect to 1. If x_0 is repelling then any orbit starting in a neighbor of x_0 will separate from the periodic orbit at each iteration with an average rate of $a^{\frac{1}{k}}$. Finally, for a non-periodic point x_0, the average rate of separation if given by $a = \lim_{k\to\infty}(|f'(x_0)| \times |f'(x_1)| \times \ldots \times |f'(x_k)|)^{\frac{1}{k}}$. The Lyapunov exponent of the orbit is defined as being the natural logarithm of the above quantity, i.e.

$$\lambda = \lim_{k\to\infty} \frac{\ln|f'(x_0)| + \ldots + \ln|f'(x_k)|}{k} \tag{13}$$

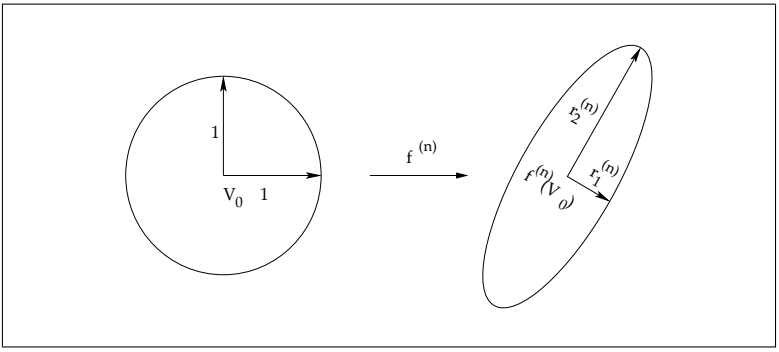


Fig. 2. Images of the unit sphere after n iterations.

For a function $f : \mathbb{R}^m \to \mathbb{R}^m$, the eigenvalues of the Jacobian matrix Df are substituted to the values of the first derivative in the above notions and we obtain m exponents.

Definition 1 *Let $f : \mathbb{R}^m \to \mathbb{R}^m$ and let e_i^n, $1 \leq i \leq m$ denote the ith greatest eigenvalue of $Df^n(x_0)$. The ith Lyapunov exponent is defined by*

$$\lambda_i = \lim_{n\to\infty} \frac{\ln e_i^n}{n}$$

f is *stable* in the vicinity of x_0 if the greatest Lyapunov exponent is a negative number. Otherwise the coefficients $\lambda_i > 0$ correspond to the unstable directions. $\lambda_i = 0$ indicates that there is a weak divergence (sub-exponential) in the ith direction. In $\mathbb{R}^m$, to compute the Lyapunov exponents of $f : \mathbb{R}^m \to \mathbb{R}^m$ is equivalent to compute the images by f of an unit sphere U with center $\boldsymbol{x_0} \in \mathbb{R}^m$ [1]. As illustrated in Figure 2, we obtain a m-dimensional ellipsoid E. The lengths of the axes of E yield information on the stability of f. In practice, the images of U may evolve in three ways:

- The images of U tend towards the simple point p. f is contracting in any direction. p is an attracting fixed point of f, or a *sink*.
- The images of U are strictly growing in all directions. f is expansive in any direction and the related fixed point is a *source*.
- U becomes a very long and thin ellipsoid. f is contracting in some directions and expansive in others. The related fixed point is a *saddle*.

So, the Lyapunov exponents compute the rates of shrinking and stretching in the vicinity of the dynamics, starting from x_0. This leads us introduce an alternative definition of the Lyapunov exponents, equivalent to Definition 1.

Definition 2 *Let $f : \mathbb{R}^m \to \mathbb{R}^m$ and let r_i^n, $1 \leq i \leq m$, denote the length of the ith longest orthogonal axis of the ellipsoid $Df^n(x_0)U$. The ith Lyapunov exponent is defined by*

$$\lambda_i = \lim_{n\to\infty} \frac{\ln r_i^n}{n}$$

We have seen that the Lyapunov exponents can be defined in two equivalent ways, by means of the eigenvalues of the Jacobian matrix, or by means of the images by f of the unit sphere. These two definitions yield two different algorithms, described in Section 5, to determine the stability of a loop. In the next section, we focus on the automatic generation of the Jacobian matrix needed by the stability test.

4 Generation of the Semantic Equations

As explained in Section 3, we determine the stability of the function f modeling the propagation of the errors inside a loop by means of the Lyapunov exponents. The exact definition of f stems from the semantics of Section 2. More precisely, the semantics is needed to build f from the syntax of the program and, next, we have to determine the Jacobian matrix Df. However, in practice, we directly build Df from the syntax of the program. The actual calculation, using Df, of the Lyapunov exponents is detailed in Section 5.

Let $r_1^{\ell_1} = f_1\boldsymbol{\varepsilon} + \sum_{\ell\in\mathcal{L}} \omega_1^{\ell}\boldsymbol{\varepsilon}_\ell$ and $r_2^{\ell_2} = f_2\boldsymbol{\varepsilon} + \sum_{\ell\in\mathcal{L}} \omega_2^{\ell}\boldsymbol{\varepsilon}_\ell$ be two floating-point numbers with errors occurring at the control points ℓ_1 and ℓ_2 of the program and let us focus on the operation $r_1^{\ell_1} \diamond^{\ell_3} r_2^{\ell_2}$. This operation sets the value $r_3^{\ell_3} = f_3\boldsymbol{\varepsilon} + \sum_{\ell\in\mathcal{L}} \omega_3^{\ell}\boldsymbol{\varepsilon}_\ell$ of point ℓ_3 to the result of $\diamond$ and can be viewed as a function of the variables $f_1, \omega_1^{\ell_1}, \ldots, \omega_1^{\ell_k}, \omega_1^{hi}, f_2, \omega_2^{\ell_2}, \ldots, \omega_2^{\ell_k}, \omega_2^{hi}$ and $f_3, \omega_3^{\ell_1}, \ldots, \omega_3^{\ell_k}, \omega_3^{hi}$, where $\mathrm{Card}(L) = k$ and $\mathcal{L}$ is the language defined in Section 2. For instance, from the equations of Figure 1, the function f_+ for an addition $r_1^{\ell_1} +^{\ell_3} r_2^{\ell_2}$ transforms the vector

$$\boldsymbol{v} = (f_1, \omega_1^{\ell_1}, \ldots, \omega_1^{\ell_k}, \omega_1^{hi}, f_2, \omega_2^{\ell_1}, \ldots, \omega_2^{\ell_k}, \omega_2^{hi}, f_3, \omega_3^{\ell_1}, \ldots, \omega_3^{\ell_k}, \omega_3^{hi})$$

into

$$\begin{aligned} f_+(\boldsymbol{v}) = \big(& f_1, \quad \omega_1^{\ell_1}, \ldots, \quad \omega_1^{\ell_3}, \ldots, \quad \omega_1^{hi}, \\ & f_2, \quad \omega_2^{\ell_1}, \ldots, \quad \omega_2^{\ell_3}, \ldots, \quad \omega_2^{hi}, \\ & \uparrow_\circ (f_1 + f_2), \omega_1^{\ell_1} + \omega_2^{\ell_1}, \ldots, \omega_1^{\ell_3} + \omega_2^{\ell_3} + \downarrow_\circ (f_1 + f_2), \ldots, \omega_1^{hi} + \omega_2^{hi}\big) \end{aligned}$$

The terms related to the operands are left unchanged and the coefficients of r_3 are assigned to the result of the addition. The Jacobian matrix of f_+ is denoted B^+ and is readily obtained from f_+. The derivative of the term $\downarrow_\circ (f_1 + f_2)$ with respect to f_1 or f_2 is $ulp(1)^2$, since for any floating-point number x we have $\downarrow_\circ (x) \approx x \cdot ulp(1)$. For instance we have:

$$B^+ = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ B^+_{\ell_3,\ell_1} & B^+_{\ell_3,\ell_2} & 0 \end{pmatrix} \tag{14}$$

with

[2] The unit in the last place (ulp) of a floating-point number is the magnitude of the least significant digit of the mantissa of a floating-point number [7].

$$B^{+}_{\ell_3,\ell_1} = \; ; \begin{array}{c} \\ f_3 \\ \omega_3^{\ell_1} \\ \vdots \\ \omega_3^{\ell_3} \\ \vdots \\ \omega_3^{\ell_n} \\ \omega_3^{\ell_{hi}} \end{array} \begin{array}{c} \begin{array}{ccccccc} \frac{\partial}{\partial f_1} & \frac{\partial}{\partial w_1^{\ell_1}} & \dots & \frac{\partial}{\partial w_1^{\ell}} & \dots & \frac{\partial}{\partial w_1^{\ell_k}} & \frac{\partial}{\partial w_1^{hi}} \end{array} \\ \begin{pmatrix} 1 & 0 & \dots & 0 & \dots & 0 & 0 \\ 0 & 1 & \ddots & & & \vdots & 0 \\ 0 & 0 & \ddots & \ddots & & \vdots & \vdots \\ ulp(1) & \vdots & \ddots & 1 & \ddots & \vdots & 0 \\ 0 & \vdots & & \ddots & \ddots & 0 & \vdots \\ 0 & \vdots & & & \ddots & 1 & 0 \\ 0 & 0 & \dots & 0 & \dots & 0 & 1 \end{pmatrix} \end{array} \tag{15}$$

$B^{+}_{\ell_3,\ell_2}$ is defined similarly and I denotes the identity block. The Jacobian matrix $B^{\times}$ for a multiplication is obtained in the same way from the function

$$\begin{aligned} f_{\times}(\boldsymbol{v}) = (\quad & f_1, \quad \omega_1^{\ell_1} \quad ,\dots, \quad \omega_1^{\ell_3} \quad ,\dots, \omega_1^{hi}, \\ & f_2, \quad \omega_2^{\ell_1} \quad ,\dots, \quad \omega_2^{\ell_3} \quad ,\dots, \omega_2^{hi}, \\ & \uparrow_{\circ}(f_1 \times f_2), f_1\omega_2^{\ell_1} + f_2\omega_1^{\ell_1}, \dots, f_1\omega_2^{\ell_3} + f_2\omega_1^{\ell_3} + \downarrow_{\circ}(f_1 \times f_2), \dots, \omega_3^{hi}) \end{aligned}$$

in which the higher-order term $\omega_3^{\ell_{hi}} = f_1\omega_2^{\ell_{hi}} + f_2\omega_1^{\ell_{hi}} + \sum_{\ell,\ell' \in \mathcal{L}^+} \omega_1^{\ell}\omega_2^{\ell'}$. The Jacobian matrix $B^{\times}$ is built is the same way to B^{+} in Equation (14). For example, we obtain the block:

$$B^{\times}_{\ell_3,\ell_1} = \; ; \begin{array}{c} \\ f_3 \\ \omega_3^{\ell_1} \\ \vdots \\ \omega_3^{\ell_3} \\ \vdots \\ \omega_3^{\ell_n} \\ \omega_3^{\ell_{hi}} \end{array} \begin{array}{c} \begin{array}{ccccccc} \frac{\partial}{\partial f_1} & \frac{\partial}{\partial w_1^{\ell_1}} & \dots & \frac{\partial}{\partial w_1^{\ell}} & \dots & \frac{\partial}{\partial w_1^{\ell_k}} & \frac{\partial}{\partial w_1^{\ell_{hi}}} \end{array} \\ \begin{pmatrix} f_2 & 0 & \dots & \dots & \dots & 0 & 0 \\ \omega_2^{\ell_1} & f_2 & \ddots & & & \vdots & 0 \\ \vdots & 0 & \ddots & \ddots & & \vdots & \vdots \\ U & \vdots & \ddots & f_2 & \ddots & \vdots & 0 \\ \vdots & \vdots & & \ddots & \ddots & 0 & \vdots \\ \omega_2^{\ell_n} & 0 & \dots & \dots & 0 & f_2 & 0 \\ \omega_2^{\ell_{hi}} & S & \dots & S & \dots & S & S' \end{pmatrix} \end{array} \tag{16}$$

where

$$U = \omega_2^{\ell_2} + f_2 ulp(1) \qquad S = \sum_{\ell \in \mathcal{L}^+} \omega_2^{\ell} \qquad S' = S + f_2$$

Given a piece of code p corresponding to the body of a loop and the language $\mathcal{L}$ built on the labels of p, the Jacobian matrix M of the semantic function defined from p is built in blocks. For example, the operation $r_1^{\ell_1} \times^{\ell_3} r_2^{\ell_2}$ makes the system fill the blocks corresponding to the partial derivative of the variables of r_3 by the variables of r_1 and r_2. Roughly speaking, if $\frac{\partial r^{\ell_3}}{\partial r^{\ell_1}}$ and $\frac{\partial r^{\ell_3}}{\partial r^{\ell_2}}$ denote blocks of the full matrix, the multiplication leads us to fill M as follows:

$$M = \begin{array}{c} \\ \vdots \\ r^{\ell_1} \\ \vdots \\ ; \; r^{\ell_2} \\ \vdots \\ r^{\ell_3} \\ \vdots \end{array} \begin{array}{c} \begin{array}{ccccccc} \cdots & \frac{\partial}{\partial \boldsymbol{r_{\ell_1}}} & \cdots & \frac{\partial}{\partial \boldsymbol{r_{\ell_2}}} & \cdots & \frac{\partial}{\partial \boldsymbol{r_{\ell_3}}} & \cdots \end{array} \\ \begin{pmatrix} . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & B^{\times}_{\ell_3,\ell_1} & . & B^{\times}_{\ell_3,\ell_2} & . & . & . \\ . & . & . & . & . & . & . \end{pmatrix} \end{array} \tag{17}$$

From a formal point of view, let $\sigma : \mathrm{Id} \to \mathcal{L}$ denote an environment $\sigma \in$Env mapping variables to labels and let $M \in$Mat denote a Jacobian matrix. For an expression e^{ℓ}, M is set to the new value $\mathcal{E}(e^{\ell}, M, \sigma)$ by the function $\mathcal{E} : \mathrm{Expr} \times \mathrm{Mat} \times \mathrm{Env} \to \mathrm{Mat}$ defined by:

$$\mathcal{E}(c^{\ell}, M, \sigma) = M$$

$$\mathcal{E}(x^{\ell}, M, \sigma) = M + I_{\ell,\sigma(x)}$$

$$\mathcal{E}\big((e_1^{\ell_1} \diamond^{\ell} \varepsilon_2^{\ell_2}), M, \sigma\big) = \mathcal{E}\big(e_2^{\ell_2}, \mathcal{E}(e_1^{\ell_1}, M, \sigma), \sigma\big) + B^{\diamond}_{\ell,\ell_1} + B^{\diamond}_{\ell,\ell_2}$$

A constant c does not modify M. I_{ℓ_1,ℓ_2} denotes the identity block set at position $\frac{\partial r^{\ell_1}}{\partial r^{\ell_2}}$. If $\sigma(x) = \ell_2$ then an occurrence of x^{ℓ_1} makes us insert the block I_{ℓ_1,ℓ_2} in the matrix M. For an operation $\diamond$, M is set as described in Equation (17). For a piece of code p related to the body of a loop, the matrix M is generated by the function $\mathcal{P} : \mathrm{Prg} \times \mathrm{Mat} \times \mathrm{Env} \to \mathrm{Mat} \times \mathrm{Env}$ (partly) defined below.

$$\mathcal{P}(x^{\ell_1} = e^{\ell_2}, M, \sigma) = \big(\mathcal{E}(e^{\ell_2}, M, \sigma) + I_{\ell_1,\ell_2} \;,\; \sigma[x \mapsto \ell_2]\big)$$

$$\mathcal{P}\;(p_1; p_2, M, \sigma) = \big(\lambda(M', \sigma').\mathcal{P}(p_2, M', \sigma')\big)\;\mathcal{P}(p_1, M, \sigma)$$

The full matrix for the body p of a loop is generated by $\mathcal{P}(p, O, [])$ where O denotes the null matrix and $[]$ the empty environment. For example, the matrix obtained for the program below is:

```
while (x! = 0){
    p : x^{ℓ4} = x^{ℓ1} ×^{ℓ3} x^{ℓ2};
}
```

$$\mathcal{P}(p, O, []) = \begin{array}{c} \\ r^{\ell_1} \\ r^{\ell_2} \\ ; \; r^{\ell_3} \\ r^{\ell_4} \end{array} \begin{array}{c} \begin{array}{cccc} \frac{\partial}{\partial \boldsymbol{r_{\ell_1}}} & \frac{\partial}{\partial \boldsymbol{r_{\ell_2}}} & \frac{\partial}{\partial \boldsymbol{r_{\ell_3}}} & \frac{\partial}{\partial \boldsymbol{r_{\ell_4}}} \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & I \\ 0 & 0 & 0 & I \\ B^{\times}_{\ell_3,\ell_1} & B^{\times}_{\ell_3,\ell_2} & 0 & 0 \\ 0 & 0 & I & 0 \end{pmatrix} \end{array}$$

5 Abstract Calculation of the Lyapunov Exponents

In this section, we present two algorithms to prove the stability of the floating-point calculations carried out inside a loop. The algorithm of Section 5.1 is based on Definition 1. It involves less calculation than the second algorithm but

only globally checks that no error term diverges. The algorithm of Section 5.2 is based on Definition 2 and precisely determines which error terms make the loop unstable.

5.1 Global Stability of a Loop

We show how to use the Jacobian matrices generated in Section 4 to determine the stability of the calculations made inside a loop. The analysis is based on an abstract interpretation of the semantics of Section 2, in which the floating-point numbers and the error terms are abstracted by intervals of floating-point numbers. Hence, if the stability of a loop is assessed, we can ensure that any actual execution based on a concretization of the abstract values is stable.

As explained in Section 3, the Lyapunov exponents are used to determine the stability of an orbit defined by the initial point $x_0 \in \mathbb{R}^m$ and the iterates of $f : \mathbb{R}^m \to \mathbb{R}^m$. From Definition 1, the exponents are computed from the eigenvalues of the Jacobian matrix $Df^n(x_0)$ of the nth iterate of f. Recall that the calculation is stable is all the exponents are negative. Our test is based on an abstract interpretation of the semantics of Section 2 in which the coefficients of the error terms are intervals. $Df^n(x_0)$ is computed as follows. Since $Df^n(x_0) = D(f(f^{n-1}(x_0)))$, we have

$$\begin{aligned} Df^n(x_0) &= Df^{n-1}(x_0) \times Df(f^{n-1}(x_0)) \\ &= Df^{n-1}(x_0) \times Df(x_{n-1}) \end{aligned}$$

Finally, by recurrence, $Df^n(x_0) = Df(x_{n-1}) \times \ldots \times Df(x_0)$. So, we focus on the product of interval matrices $Df^n(x_0) = Df(x_{n-1}) \times \ldots \times Df(x_0)$ whose eigenvalues indicate the stability of the first n iterates. This approach leads us to evaluate (by abstract interpretation) the first n iterates of the loop in order to compute $x_0,\ x_1 = f(x_0), \ldots, x_n$. At each step k, the matrix Df, generated from the syntax of the program by the method of Section 4 and which is composed of formal expressions, is valued, yielding $Df(x_k)$, and the new product $Df^{k+1}(x_0) = Df(x_k) \times \ldots \times Df(x_0)$ is computed.

$Df^k(x_0)$ is a matrix of intervals and we use the Gerschgorin disks [5,9] to find an upper bound of the greatest module of its eigenvalues, as explained below. For a matrix A of size $m \times m$, the eigenvalues are contained in $D_1 \cap D_2$, where D_1 and D_2 are the disks of the complex plane $\mathbb{C}$ defined by:

$$D_1 = \cup_{1 \le i \le m} D_{1,i} \qquad\qquad D_2 = \cup_{1 \le j \le m} D_{2,j}$$

$D_{1,i}$ is the disk of center a_{ii} and radius $r_{1,i} = \sum_{1 \le j \le m, j \neq} |a_{ij}|$. Similarly, $D_{2,j}$ is the disk of center a_{jj} and radius $r_{2,j} = \sum_{1 \le i \le m, i \neq} |a_{ij}|$. So, an upper bound of the module of the greatest eigenvalues of A is given by:

$$G(A) = \max\big(|a_{ii}| + \max(r_{1,i}, r_{2,i}) \ : \ 1 \le i \le m\big)$$

From Definition 1, the greatest Lyapunov exponent after k iterations is bounded by the abstract Lyapunov exponent:

$$\lambda_k^\sharp = \frac{\ln\big(G(Df^k(x_0))\big)}{k} \tag{18}$$

Let k_0 be the first iteration of the algorithm such that $\lambda^\sharp < 0$, let f denote the semantic function of the body of the loop and let f^k be the function f iterated k times. The algorithm stops after k_0 iterations, i.e. as soon as $\lambda^\sharp < 0$. This ensures that f^{k_0} is a contracting function. Consequently, $f^{k_0}(x_0) \prec x_0$, where $\prec$ is the componentwise ordering used for the abstract domain of Section 2 [13]. So, any error term is decreasing after k_0 iterations and $f^{k_0}(f^{k_0}(x_0)) \prec f^{k_0}(x_0) \prec x_{k_0}$, etc. Note that this condition does not imply that the error terms are always decreasing between the iterations nk_0 and $(n+1)k_0$ for any integer n. However, it implies that no error term significantly diverges since the function becomes contracting after some more iterations. Obviously, one can also force the algorithm to stop after a predefined number of iterations k_{max} and return the conservative result that the loop is unstable if the termination condition is not satisfied. To sum up, the algorithm we use is given below.

while $\lambda^\sharp >= 0$ do
1. Unfold the loop once: $\rho^\sharp = [\![p]\!]^\sharp\ \rho^\sharp$; $k = k + 1$
2. Valuate Df: $\Delta = Df(\rho^\sharp)$
3. Compute the new Jacobian Df^k: $Df^k = \Delta \times Df^{k-1}$
4. Approximate the greatest eigenvalue: $e = \mathrm{G}(Df^k)$
5. Compute the Lyapunov exponent after k iterations: $\lambda^\sharp = \frac{\ln(e)}{k}$

We end this section by showing the results obtained with an implementation of this method. The top two curves of Figure 3 correspond to the loop whose body is `x = x*x`. The x-axis corresponds to the iterations k, while the y-axis corresponds to the upper approximation $\lambda_k^\sharp$ of the Lyapunov exponents. The first case is stable, since the initial value of `x` is $[0.0, 0.95] + [0.0, 0.01]\varepsilon_x$. $\lambda_k^\sharp$ becomes negative after a number of iterations, indicating that the loop is stable. In the second case we have `x`$= [0.0, 0.95] + [0.0, 0.1]\varepsilon_x$. Even if the calculation carried out with floating-point numbers is stable and converges to 0, the initial real number can be as great as 1.05 and the errors made during the execution of the loop may be unbounded. We can see on the top right curve of Figure 3 that $\lambda_k^\sharp$ is positive and increasing. Our second example concerns a classical numerical algorithm, Jacobi's iterative method to solve a system of linear equations. We consider the systems:

$$(S_1)\ :\ \begin{cases} 2x + 3y = 3 \\ 4x + \frac{3}{2}y = 3 \end{cases} \qquad (S_2)\ :\ \begin{cases} 2x + y = \frac{5}{2} \\ x + 3y = \frac{5}{2} \end{cases}$$

To solve (S_1) and (S_2) by Jacobi's method [8], the following sequences must be computed:

$$(S_1)\ :\ \begin{cases} x_{n+1} = \frac{3}{2} - \frac{3}{2}y_n \\ y_{n+1} = 2 - \frac{8}{3}x_n \end{cases} \qquad (S_2)\ :\ \begin{cases} x_{n+1} = \frac{5}{6} - \frac{1}{2}y_n \\ y_{n+1} = \frac{5}{6} - \frac{1}{3}x_n \end{cases}$$

The solution to both (S_1) and (S_2) is $x = \frac{1}{2}$ and $y = \frac{2}{3}$, but (S_1) is unstable for Jacobi's method while (S_2) is stable[3]. Indeed, a C program implementing (S_1) in

[3] A sufficient condition to the stability of Jacobi's method for a linear system of equations is that $|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}|$, where the a_{ij} are the coefficients of the variables [8].

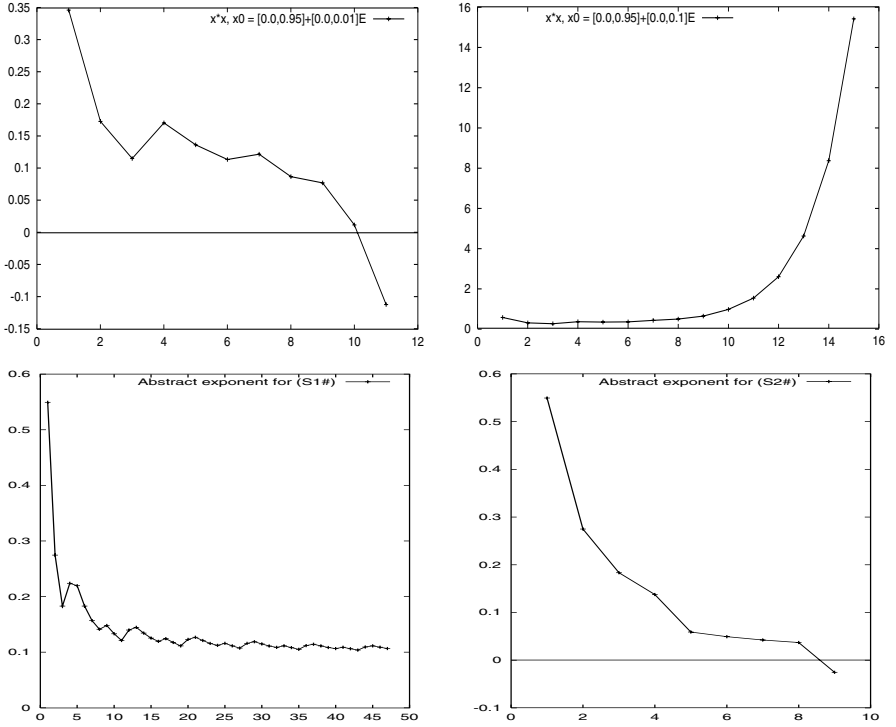


Fig. 3. Exponents $\lambda^\sharp$ computed by our implementation of the first algorithm for the examples described in the text.

double precision, compiled with GCC and executed on a Pentium III processor yields after 15 iterations $x = 19408738461538.50$, $y = 13275241025642.0$ if $x_0 = y_0 = 3$. Under the same assumptions, our implementation of (S_2) yields after 15 iterations $x = 0.500001$ and $y = 0.666668$. The bottom curves of Figure 3 show the values of the abstract Lyapunov exponents computed by our implementation for the systems:

$$(S_1^\sharp) \ : \ \begin{cases} x_{n+1} = [1.2, 1.6] - [1.2, 1.6]y_n \\ y_{n+1} = [2.0, 2.5] - [2.5, 3.0]x_n \end{cases} \qquad (S_2^\sharp) \ : \ \begin{cases} x_{n+1} = [0.80, 0.85] - [0.4, 0.6]y_n \\ y_{n+1} = [0.80, 0.85] - [0.30, 0.35]x_n \end{cases}$$

The initial values are $x_0^\sharp = y_0^\sharp = [2.0, 3.0]$. The curve corresponding to $(S_2^\sharp)$ becomes negative after a number of iterations, indicating that any actual execution of this program using concrete values in the given intervals is stable. The curve related to the unstable system $(S_1^\sharp)$ remains positive, which means that the calculations carried out in the program are unstable. Finally, let us remark that, for (S_2), the function evaluated in the loop is a linear map defined by the matrix

$M = \begin{pmatrix} \frac{5}{6} & \frac{-1}{2} \\ \frac{5}{6} & \frac{-1}{3} \end{pmatrix}$ and is stable. However, $G(M) = \frac{5}{3}$ and the technique introduced in ref. [9] to determine the stability of a linear loop by the test $G(M) < 1$ also requires the unfolding of the loop. The abstract Lyapunov exponent test $\lambda^\sharp < 0$ is also verified after some iterations.

5.2 Calculation of Every Exponent

The algorithm of Section 5.1 enables the global determination of the stability of the calculations carried out inside a loop. However, if the loop is unstable, it does not help to understand the source of the imprecision, i.e. the error terms which do not converge. In this section, we introduce a second algorithm, more complex than the previous one, to compute the full sequence of Lyapunov exponents.

Let f be the semantic function related to the body of a loop. By definition 2, we directly compute the images by f of the unit sphere U [1]. Again, the calculation are based on an abstract version of the semantics in which the coefficients of the series are intervals. After k iterations, we obtain an ellipsoid E_k whose axes indicate the contracting or expanding nature of f in a given direction. More precisely, we aim at computing $Df^k(x_0)U = Df(x_{k-1}) \dots Df(x_0)U$. Let $(e_1^0, e_2^0, \dots, e_m^0)$ be an orthonormal basis of $\mathbb{R}^m$, we compute

$$z_1 = Df(x_0)e_1^0, \dots, z_m = Df(x_0)e_m^0 \tag{19}$$

The vectors $(z_1, \dots, z_m)$ are axes of the ellipsoid $Df(x_0)U$ but they are not necessarily orthogonal. So, the next step consists of computing a new set $(y_1, \dots, y_m)$ of orthogonal vectors which define an ellipsoid with the same volume as $Df(x_0)U$. In the current implementation, this is done by the Gram-Schmidt procedure [15] but more sophisticated algorithms shXould be used. The next iteration is carried out with the new set of vectors

$$e_1^1 = y_1, e_2^1 = y_2, \dots, e_m^1 = y_m \tag{20}$$

and, after k iterations, we obtain an ellipsoid whose axes are $(e_1^k, e_2^k, \dots, e_m^k)$. Following the definitions of Section 3, the total expansion r_i^k in the ith direction is given by e_i^k and the related abstract Lyapunov exponent is $\lambda_i^\sharp = \frac{\ln r_i^k}{k}$. Finally, since the ellipsoid may have long and short axes after some iterations, we normalize the basis at each step. Equation (20) becomes

$$e_1^k = \frac{y_1^k}{\|y_1^k\|}, \dots, e_m^k = \frac{y_m^k}{\|y_m^k\|} \tag{21}$$

where y_i^k denotes the vector y_i at iteration k. The ith axis r_i^k of the ellipsoid after k iterations has length $r_i^k = \|y_i^k\| \dots \|y_i^1\|$, so the ith abstract Lyapunov exponent after k iterations is

$$\lambda_i^{\sharp k} = \frac{\ln\|y_i^k\| + \dots + \ln\|y_i^1\|}{k} \tag{22}$$

The algorithm is summarized below. The termination is managed in the same way as for the first algorithm of Section 5.1.

let E be an orthonormal basis of $\mathbb{R}^m$
while $\exists i,\ 1 \leq i \leq m\ :\ \lambda_i >= 0$ do
1. Unfold the loop once: $\rho^\sharp = [\![p]\!]^\sharp\ \rho^\sharp$; $k = k + 1$
2. Valuate Df: $\Delta = Df(\rho^\sharp)$
3. Compute the axes Z of the new ellipsoid: $Z = \Delta \times E$
4. Orthogonalize Z: $Y = (y_1, \ldots, y_m) = \text{Gram-Schmidt}(Z)$
5. Normalize Y: $E = (\frac{y_1}{\|y_1\|}, \ldots, \frac{y_m}{\|y_m\|})$
6. For $1 \leq i \leq m$, compute the new exponents by Equation (22), using Y

We end this section by presenting some experimental results provided by an implementation of this algorithm. The top two curves of Figure 4 show the values of the abstract Lyapunov exponents at each iteration for a loop computing `x=x*x`. The top left curve corresponds to a stable case in which the initial abstract value of `x` is $[0.0, 0.95] + [0.0, 0.01]\varepsilon_x$. The values of the abstract exponents related to the first-order error on `x` and to the higher-order error on `x` are plotted and we can observe that both become negative after a number of iterations, indicating that the program is stable in this case. The top right curve corresponds to an unstable case for the same program since the initial value of `x` is $[0.90, 0.95] + [0.0, 0.1]\varepsilon_x$. Here, the only value of the abstract exponent related to the first-order error term becomes negative after a number of iterations. The algorithm detects that there is an instability due to the the higher-order error on `x`, as predicted by the theory (see Section 3). Our second example concerns the system of linear differential equations with initial conditions:

$$(S)\ :\ \begin{cases} \dot{x} &= -2x \\ \dot{y} &= x + 3y \end{cases} \qquad x(0) = 1.0, \quad y(0) = 2.0 \tag{23}$$

The exact solution is $x(t) = x(0)e^{-2t}$, $y(t) = -\frac{1}{5}x(0)e^{-2t} + (y(0) - x(0))e^{3t}$. The solution $x(t)$ is insensitive to a small variation of $x(0)$ since this term is multiplied by e^{-2t} but $y(t)$ is sensitive to $x(0)$ and $y(0)$. The algorithm introduced in this section enables to detect this phenomenon in a program computing a numerical solution to (S) by Euler's method [15]. Let us consider the sequence:

$$\begin{cases} x_{n+1} &= x_n - 2hx_n \\ y_{n+1} &= y_n + hx_n + 3hy_n \end{cases} \qquad x_0 = 1.0, \quad y_0 = 2.0 \tag{24}$$

where the step $h = 0.1$. For the abstract initial condition $x_0 = [1.0, 1.0] + [0.0, 0.1]\varepsilon_{x_0}$ and $y_0 = [2.0, 2.0] + [0.0, 0.1]\varepsilon_{y_0}$, the abstract Lyapunov exponents computed by our algorithm are given in Figure 4. The left bottom curve shows the values of the exponent related to the first-order error on x_n. Since it is negative, we conclude that the sequence $(x_n)_n$ is stable. Conversely, the values of the exponent related to the first-order error on y_n, given in the right bottom curve, are positive, indicating that this calculation is unstable.

6 Conclusion

In this article, we introduced a static test to determine whether the calculations carried out in a loop are stable. This test is based on an abstract calculation of

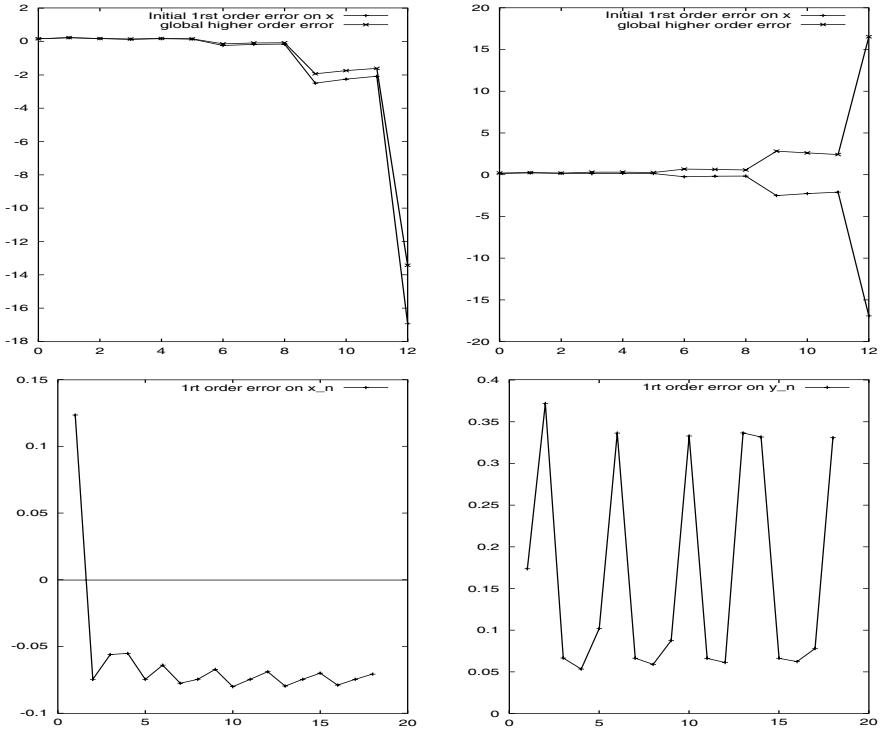


Fig. 4. Exponents $\lambda^\sharp$ computed by our implementation of the second algorithm for the examples described in the text.

the Lyapunov exponents for the semantic function defined by the body of a loop and the semantics of floating-point numbers with errors [9,13]. In Section 5 we introduced two algorithms, to test that a loop is globally stable and to determine which operations of a loop are unstable. We used classic numerical methods to demonstrate the usefulness of both algorithms.

A first advantage of our approach is that non-linear terms as well as higher-order error terms are fully analyzed. Not to accept non-linear terms, as proposed by some (dynamic) approaches [12,16], is an important restriction on the accepted codes and to neglect higher-order errors may lead to erroneous conclusions. A second advantage is that the stability test does not depend on parameters of the analysis. The point is that, with the usual widening operators, to precisely analyze even a stable calculation (for example the system (S_2) of Section 5.1) the loop must be unfolded a finite but unknown number of times and the unfolded code must be executed another finite but unknown number of times before widening [10]. So, if a loop is declared unstable by the analyzer, there is no way of knowing whether this is due to a bad parameter value or to an actual numerical problem. The evolution of the abstract Lyapunov exponent, as shown in the figures 3 and 4 yields far more information.

The current implementation, described in Section 5 could be significantly improved if more elaborate numerical algorithms were used. The matrices generated by the method of Section 4 contain few non-zero elements and a sparse representation would be well-suited. For the second algorithms, a more accurate orthogonalization procedure should be used. Finally, we are investigating other ways to improve the treatment of loops in the analyzer described in ref. [10]. We believe that by mixing classic loop unfolding with a dynamic repartitioning of the control points, we could assess the stability of many new programs. This technique should complement the one introduced in this article, each being better-suited to some kinds of codes.

References

1. K. T. Alligood, T. D. Sauer, and J. A. Yorke. *Chaos, an Introduction to Dynamical Systems.* Springer-Verlag, 1996.
2. ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*, std 754-1985 edition, 1985.
3. M. F. Barnsley. *Fractals Everywhere, Second Edition.* Academic Press, 1993.
4. A. F. Beardon. *Iteration of Rational Functions.* Number 132 in Graduate Texts in Mathematics. Springer-Verlag, 1991.
5. F. Chatelin. *Valeurs propres des matrices.* Masson, 1988.
6. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Symbolic Computation*, 2(4):511–547, 1992.
7. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1), 1991.
8. G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 2d edition, 1990.
9. E. Goubault. Static analyses of the precision of floating-point operations. In *Static Analysis Symposium, SAS'01*, number 2126 in Lecture Notes in Computer Science. Springer-Verlag, 2001.
10. E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: a simple abstract interpreter. In *European Symposium on Programming, ESOP'02*, number 2305 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
11. D. Knuth. *The Art of Computer Programming - Seminumerical Algorithms.* Addison Wesley, 1973.
12. P. Langlois and F. Nativel. Improving automatic reduction of round-off errors. In *IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, volume 2, 1997.
13. M. Martel. Propagation of roundoff errors in finite precision computations: a semantics approach. In *European Symposium on Programming, ESOP'02*, number 2305 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
14. F. C. Moon. *Chaotic and Fractal Dynamics.* Wiley-Interscience, 1992.
15. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 1992.
16. J. Vignes. A survey of the CESTAC method. In *Proceedings of Real Numbers and Computer Conference*, 1996.

On Termination of Logic Programs with Floating Point Computations

Alexander Serebrenik and Danny De Schreye

Department of Computer Science, K.U. Leuven
Celestijnenlaan 200A, B-3001, Heverlee, Belgium
{Alexander.Serebrenik, Danny.DeSchreye}@cs.kuleuven.ac.be

Abstract. Numerical computations form an essential part of almost any real-world program. Traditional approaches to termination of logic programs are restricted to domains isomorphic to $\mathcal{N}$, more recent works study termination of integer computations. Termination of computations involving real numbers is cumbersome and counter-intuitive due to rounding errors and implementation conventions. We present a novel technique that allows us to prove termination of such computations. Our approach extends the previous work on termination of integer computations.

1 Introduction

Numerical computations form an essential part of almost any real-world program. Clearly, in order for a termination analyser to be of practical use it should contain a mechanism for proving termination of such computations. However, this topic attracted less attention of the research community. Most of the approaches suggested so far were restricted to programs depending on integer computations only [13,23] or computations over domains isomorphic to the set of natural numbers $\mathcal{N}$ [20]. The reason for doing so is that, as shown by Dershowitz *et al.* [13], termination of computations with real numbers, may be counter-intuitive, i.e., the actually observed behaviour does not necessary coincide with the theoretically expected one[1]. Difficulties encountered while working with the implementations of real numbers explain why some Prolog implementations, such as Open Prolog [7], do not include real-numbers arithmetics.

The reason for these difficulties is that the real numbers represented by computer are not mathematical objects, but their floating-point approximations.

Example 1. Consider the following programs.

$$(P_1)\ p(X) \leftarrow X > 0, X1 \ is\ X * 0.25, p(X1).$$
$$(P_2)\ q(X) \leftarrow X > 0, X1 \ is\ X * 0.75, q(X1).$$
$$(P_3)\ r(X) \leftarrow X > 0, X1 \ is\ X - 0.25, r(X1).$$

[1] Examples provided in [13] contain cuts; however, similar behaviour can be observed if cut is excluded from the language.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 151–164, 2002.
© Springer-Verlag Berlin Heidelberg 2002

If the numbers considered were real numbers, $p(1)$ and $q(1)$ would not terminate with respect to P_1 and P_2, respectively, while $r(1)$ would terminate with respect to P_3. However, computers do not work with the reals, but with the floating point numbers and computations are imprecise due to the rounding. For example, SICStus [25] rounds $3 \times 10^{15} - 0.25$ to 3×10^{15}, resulting in non-termination of $r(3 \times 10^{15})$ with respect to P_3.

Despite the similarity between P_1 and P_2 the termination behaviour of $p(1)$ and $q(1)$ with respect to the programs is completely different. That is, $p(1)$ terminates with respect P_1 and $q(1)$ may not terminate with respect to P_2. The reason is that if rounding is done to the nearest value, then for some t, $t * 0.75$ can be rounded upwards to t. On the other hand, for all s, $s * 0.25$ can never be rounded to s, since 0 is always closer to $s * 0.25$ than s.

The rounding to the nearest policy is the default rounding policy. However, a user can specify the rounding to be done, for example, toward zero. In this case, both $p(1)$ terminates with respect to P_1 and $q(1)$ terminates with respect to P_2. □

Example 1 shows that termination depends on the domain of the computation. However, in the early days of computing the domain of the floating point numbers was completely depending on the actual implementation, making the analysis almost impossible. To solve this problem a number of standards [1,2,17] were suggested. As the following example hints, the existing situation, discussed in Section 3, is still not free from anomalies.

Example 2. As we have seen in the previous example, if the rounding to the nearest value is assumed, $q(1)$ does not terminate with respect to P_2. The following program P_4 is logically equivalent to P_2.

$$(P_4)\ q(X) \leftarrow 1 + X > 1, X1 \ \mathit{is}\ X * 0.75, q(X1).$$

In general, one might expect $1 + X > 1$ to be equivalent to $X > 0$ and $q(1)$ to be non-terminating with respect to P_4 in the same way as it does not terminate with respect to P_2. However, $q(1)$ turns out to be terminating with respect to P_4!

The reason for this is that the precision of floating-point numbers near 0 is much higher than near 1. Thus, there exist real numbers r, such that $r > 0$ holds, but $1+r$ is computed to be equal to 1. When such a number is computed, the test $1 + X > 1$ fails and the computation terminates. □

Even more surprising behaviour can be observed if non-numerical values are considered. Those are required by the standards to make the arithmetic functions total. For example, $1.0/0.0$ is defined to be $+\infty$, while $0.0/0.0$ to be *nan*. Moreover, zero is required to be signed, meaning that there should be two values representing zero: 0.0 and -0.0. The values should be numerically equal, that is, $0.0 =:= -0.0$ should hold. As the following example illustrates, these values can result in an extremely contrived behaviour.

Example 3. Consider the following program:

$$r(X) \leftarrow Y \textit{ is } 1/X, s(Y).$$
$$s(X) \leftarrow X > 0, X1 \textit{ is } X + 1, s(X1).$$

We would like to study execution of $r(0.0)$ and of $r(-0.0)$ with respect to this program. Some Prolog implementations, such as the MasterProLog [18] and the *iso*-mode of SICStus Prolog [25] report an error upon zero division.

On the other hand, most of the Prolog implementations (CIAO Prolog [8], ECL^{i}PS^e [3] and the default mode of SICStus Prolog [25]) produce $s(+\infty)$ for $r(0.0)$ and $s(-\infty)$ for $r(-0.0)$. The order relationship $>$ is defined on the infinity values to satisfy $+\infty > r > -\infty$ for all floating-point numbers r. Computing $s(+\infty)$, thus, results in non-termination, while computing $s(-\infty)$ terminates. Thus, surprisingly, there are two values t_1 and t_2, such that $t_1 =:= t_2$, $r(t_1)$ terminates with respect to this program and $r(t_2)$ does not terminate with respect to it. □

Despite of these anomalies we are going to base our termination analysis, presented in Sections 4 and 5, on the standards discussed.

2 Preliminaries

We follow the standard notation for terms and atoms. A *query* is a finite sequence of atoms. Predicate occurring in an atom A is denoted $rel(A)$. $Atom_P$ is a set of all atoms that can be constructed from the language underlying P, B_P^E is a quotient set of $Atom_P$ modulo the variant relation. An SLD-tree constructed using the left-to-right selection rule is called an LD-tree. A query Q *LD-terminates* for a program P, if the LD-tree for (P, Q) is finite.

The following definition is similar to Definition 6.30 [5].

Definition 1. *Let P be a program and p, q be predicates occurring in it. We say that*

- p refers to q in P *if there is a clause in P that uses p in its head and q in its body.*
- p depends on q in P *and write $p \sqsupseteq q$, if (p, q) is in the transitive closure of the relation* refers to.
- p and q are mutually recursive *and write $p \simeq q$, if $p \sqsupseteq q$ and $q \sqsupseteq p$.*

We recall some basic notions, related to termination analysis with respect to sets of queries. A *level mapping* is a function $|\cdot|: B_P^E \to \mathcal{N}$. The *call set* [11], $Call(P, S)$, is the set of all atoms A, such that a variant of A is a selected atom in some branch of the LD-tree for $P \cup \{\leftarrow Q\}$, for some $Q \in S$. The following definition [24] generalises the notion of acceptability with respect to a set [10, 11] by extending it to mutual recursion.

Definition 2. *Let S be a set of atomic queries and P a definite program. P is* acceptable with respect to S *if there exists a level mapping $|\cdot|$ such that*

- *for any* $A \in Call(P, S)$
- *for any clause* $A' \leftarrow B_1, \ldots, B_n$ *in* P*, such that* $\mathrm{mgu}(A, A') = \theta$ *exists,*
- *for any atom* B_i*, such that* $rel(B_i) \simeq rel(A)$ *and for any c.a.s.* σ *for* $\leftarrow (B_1, \ldots, B_{i-1})\theta$ *holds that*

$$\mid A \mid > \mid B_i\theta\sigma \mid .$$

It was shown in [10] that a program is acceptable with respect to a set if and only if it is LD-terminating for all queries in the set.

While studying computations relaying on the reals, finding a level mapping that satisfies Definition 2 might be difficult. Thus, a notion of the level mapping is extended to mappings to non-negative real numbers: $|\cdot|: B_P^E \to \mathcal{R}^+$. However, the usual acceptability decrease is insufficient to ensure termination, since the reals are not well-founded. In order to obtain the well-foundedness, and hence to prove termination, in the definition above we should require the existence of $\varepsilon > 0$ such that for any A, $A' \leftarrow B_1, \ldots, B_n$, B_i and σ as above holds that $\mid A \mid \; - \; \mid B_i\theta\sigma \mid \geq \varepsilon$. This condition is equivalent to Definition 2. Indeed, if there is a level-mapping, satisfying Definition 2, ε can be taken equal to 1. On the other hand, if there exist a mapping to non-negative real numbers $|\cdot|$ and $\varepsilon > 0$ that satisfy the condition above then $|\cdot|_2$ defined as $|t|_2 = \left\lfloor \frac{|t|}{\varepsilon} \right\rfloor$ satisfies the acceptability condition. Thus, instead of using the acceptability condition of Definition 2, we apply the equivalent condition, which we also call *acceptability.*

In [23] we extended the previous work on a constraints based approach to termination [12] to include integer computations. The main difficulty that has to be solved was the non well-foundedness of integers. We illustrate the approach proposed by means of example and refer the reader to [23] for further details.

Example 4. We are interested in proving termination of the set of queries $\{p(n) \mid n \text{ is an integer}\}$ with respect to the following program:

$$p(X) \leftarrow X > 1, X < 100, X1 \; is \; -X^2, p(X1).$$
$$p(X) \leftarrow X < -1, X > -100, X1 \; is \; X^2, p(X1).$$

Let \$1 denote the first argument. The first clause is applicable if the constraint $c_1 \equiv 1 < \$1 < 100$ holds for $p(X)$, the second one, if $c_2 \equiv -100 < \$1 < -1$ holds for $p(X)$. Thus, termination of $p(X)$ for $c_3 \equiv (\$1 \leq -100 \vee -1 \leq \$1 \leq 1 \vee \$1 \geq 100)$ is trivial. Moreover, if c_1 holds and the first clause is applied, then either c_2 or c_3 hold for the recursive call. We use this observation and specialise the program with respect to c_1, c_2 and c_3 (cf. [28]), that are called *adornments* and the program transformation is called *adorning.* In general, adornments are constructed as conjunctions of (disjunctions of) comparisons. The following program is obtained:

$$p^{c_1}(X) \leftarrow X > 1, X < 100, X1 \; is \; -X^2, -100 < X1, X1 < -1, p^{c_2}(X1).$$
$$p^{c_1}(X) \leftarrow X > 1, X < 100, X1 \; is \; -X^2,$$
$$(X1 \leq -100; (-1 \leq X1, X1 \leq 1); X1 \geq 100), p^{c_3}(X1).$$
$$p^{c_2}(X) \leftarrow X < -1, X > -100, X1 \; is \; X^2, 1 < X1, X1 < 100, p^{c_1}(X1).$$

$$p^{c_2}(X) \leftarrow X < -1, X > -100, X1 \ is \ X^2,$$
$$(X1 \leq -100; (-1 \leq X1, X1 \leq 1); X1 \geq 100), p^{c_3}(X1).$$

At the next step three *natural* level mappings are defined. In general, $\mid p^{a_1 \wedge \ldots \wedge a_n}(t_1, \ldots, t_n) \mid$ is a weighted sum of $\mid p^{a_i}(t_1, \ldots, t_n) \mid$'s. If a_i is a disjunction the corresponding level mapping is zero, otherwise $a_i = E_1 \rho E_2$ for some expressions E_1, E_2 and $\rho \in \{>, \geq\}$. Then $\mid p^{a_i}(t_1, \ldots, t_n) \mid$ is $(E_1 - E_2)(t_1, \ldots, t_n)$ if $E_1(t_1, \ldots, t_n) \ \rho \ E_2(t_1, \ldots, t_n)$ and 0 otherwise. In particular, if $a_i = \$1 > 0$, $\mid p^{a_i}(t_1, \ldots, t_n) \mid = t_1$.

In our case, the natural level mappings are $\mid p^{c_1}(n) \mid = 100 - n$ if $1 < n < 100$ and 0 otherwise; $\mid p^{c_2}(n) \mid = 100 + n$ if $-100 < n < -1$ and 0 otherwise, and $\mid p^{c_3} \mid = 0$. Acceptability of the transformed program with respect to $\{p^{c_1}(n) \mid 1 < n < 100\} \cup \{p^{c_2}(n) \mid -100 < n < -1\}$ via the specified level mappings can be easily verified, implying termination of the transformed program and, thus, of the original one. □

3 Standardisation of Arithmetics

The aforementioned standards [1,2] define a set of floating-point values, F, as a finite subset of the set of real numbers $\mathcal{R}$, characterised by five parameters: the "base" of the number system, also called the *radix*, $r \in \mathcal{N}$; the number of radix digits provided by F, also called the *precision*, $p \in \mathcal{N}$; the smallest and the largest exponents of F, $emin \in \mathcal{Z}$ and $emax \in \mathcal{Z}$, where $\mathcal{Z}$ denotes the set of integers, and a boolean flag *denorm* being *true* if F contains denormalised numbers (see further).

Given specific values for the parameters, the set of *normalised* floating point values F_N is $\{0, \pm i * r^{e-p} \mid i, e \in \mathcal{Z}, r^{p-1} \leq i \leq r^p - 1, emin \leq e \leq emax\}$ and the set of *denormalised* floating point values F_D is $\{\pm i * r^{emin-p} \mid i \in \mathcal{Z}, 1 \leq i \leq r^{p-1} - 1\}$. The main reason for introducing denormalised numbers in [17] is that the gap between zero and the smallest normalised number (r^{emin-1}) is significantly bigger than the gap between the smallest normalised number and the second smallest one ($r^{emin-1} + r^{emin-p}$). This gap near 0 resulted in a number of anomalies, such as $x = y$ being not equivalent with $x - y = 0$ or $1 \times x$ being different from x. However, the use of denormalised numbers implies much higher precision near 0 than near 1, resulting in non-equivalence of $x > 0$ and $1 + x > 1$, as observed in Example 2. Thus, the ISO/IEC standard for Language Independent Arithmetic (ISO/IEC 10967) [1] leaves the decision whether the denormalised numbers should be implemented to the system developer. Formally, the set F is defined as $F_N \cup F_D$, if $denorm = true$, and as F_N, otherwise.

For convenience, the following notions are defined: the minimal normalised number $fmin_N$ ($= r^{emin-1}$), the minimal denormalised number $fmin_D$ ($= r^{emin-p}$) and F^*, the set of floating-point values extended beyond the maximal normalised value.

These parameters shall and should satisfy the following:

$$\text{shall} \ \ 0 < r^{1-p} < 0.5, \ \ 0 < fmin_N \leq 0.5, \ \ 0 < fmin_D \leq 0.25 \qquad (1)$$

$$\text{should } 0 < r^{1-p} < 10^{-6}, \;\; 0 < \mathit{fmin}_N \leq 10^{-12}, \;\; 0 < \mathit{fmin}_D \leq 10^{-18} \tag{2}$$

Using (2) instead of (1) can be justified by the observation that the majority of the implementations meet (2) and, since variable domains allowed by (2) are significantly smaller, precision of the analysis can be improved.

Example 5. Traditionally, numbers are represented in the binary format, i.e., $r = 2$. The IEEE Standard 754 requires for the *single* format $p = 24$, $\mathit{emin} = -126$ and $\mathit{emax} = 127$, $\mathit{denorm} = \mathit{true}$. This implies that $\mathit{fmin}_N \approx 1.2 \times 10^{-38}$ and $\mathit{fmin}_D \approx 1.4 \times 10^{-45}$. Both (1) and (2) are satisfied. □

Operations on floating point numbers are defined as follows. First, the exact value is computed. In case of addition information might be lost at this step. We assume this doesn't happen, since neither [1] nor [2] specify any requirements on the precision loss at this step, making the analysis impossible. For other arithmetic functions information loss is explicitly prohibited by the standards. Then, the floating point rounding function rnd_F is applied. Finally, the boundaries of F are checked and an appropriate exception (*float_overflow* or *underflow*) is thrown if needed. Different standards specify differently whether the exception may be trapped. IEEE 754 leaves the decision to a higher-level software (for example, a Prolog system). On the other hand, ISO 10967 and ISO 13211 leave this decision to a programmer, thus, explicitly prohibiting Prolog systems from trapping exceptions as a default policy. Example 3 illustrates that most of the Prolog systems violate the latter requirement. In this paper we assume that exceptions are not trapped.

Function rnd_F maps the reals to F^* and shall satisfy for all $x \in \mathcal{R}$ $\mathit{rnd}_F(-x) = -\mathit{rnd}_F(x)$ [1]. From the four rounding modes discussed in [17]: to the nearest, toward 0, toward $-\infty$ and toward $+\infty$, the two last ones do not satisfy it. This one of the discrepancies between the standards despite the fact that both of them were adopted by the same institution (IEC). Thus, we restrict our attention only to the first two modes.

The rounding function shall satisfy for all $x \in \mathcal{R}$

$$|\,\mathit{rnd}_F(x) - x\,| \leq \mathit{rnd\_error} * r^{e_F(\mathit{rnd}_F(x)) - p}, \tag{3}$$

where $e_F(x)$ is defined as $\lfloor \log_r |\,x\,| \,\rfloor + 1$ if $|\,x\,| \geq \mathit{fmin}_N$ and as emin, otherwise. The requirement that rnd_F be a rounding function implies that $\mathit{rnd\_error} \leq 1$.

Depending on the fact whether rounding is done toward zero or to the nearest value, the rounding function rnd_F shall satisfy

$$|\,\mathit{rnd}_F(x)\,| \leq |\,x\,| \qquad \text{rounding is done toward zero,} \tag{4}$$

$$|\,\mathit{rnd}_F(x) - x\,| \leq 0.5 r^{e_F(x) - p} \text{ rounding is done to the nearest value.} \tag{5}$$

4 Examples

In this section we use the formulae of Section 3 to study the termination behaviour of a number of examples. We start by programs generalising Example 1 and then proceed to a more realistic example.

Example 6. The following program is motivated by P_1 and P_2 from Example 1. We would like to find the values of $0 < \alpha < 1$ such that $p(t)$ terminates for all floating point numbers t.

$$p(X) \leftarrow X > 0, X1 \ is \ X * \alpha, p(X1).$$

To achieve our goal we try to prove termination of $p(t)$ and collect the constraints on α we need to assume. Obviously, $p(t)$ terminates for all $t \leq 0$. Thus, in the remainder we assume $t > 0$.

At the first step, the program is transformed similarly to [23]. The following program is obtained:

$$p^{\$1>0}(X) \leftarrow X > 0, X1 \ is \ X * \alpha, X1 > 0, p^{\$1>0}(X1).$$
$$p^{\$1>0}(X) \leftarrow X > 0, X1 \ is \ X * \alpha, X1 \leq 0, p^{\$1\leq 0}(X1).$$

Next, a level-mapping is chosen to be $|\,p(t)\,| = t$. The level-mapping is well-defined, since $t > 0$ and $\alpha > 0$ are assumed. If $p(t)$ is unified with the head of the clause, the value to be stored in $X1$ is either $rnd_F(t\alpha)$, *floating_overflow* or *underflow*. If one of the later cases occurs execution terminates. Thus, we restrict our attention only to the case when the value stored in $X1$ is $rnd_F(t\alpha)$. If we can find $\varepsilon > 0$, such that for any t, $t - rnd_F(t\alpha) \geq \varepsilon$, the program will be acceptable with respect to $\{p(1)\}$.

Assume that rnd_F has the *round to nearest* property. Condition (5) implies $|\ rnd_F(t\alpha) - t\alpha\ | \leq 0.5r^{e_F(t\alpha)-p}$. We distinguish between the following cases:

- $|\ \ t\alpha\ | < fmin_N$. Then, $e_F(t\alpha) = emin$ and $r^{emin-p} = fmin_D$. Thus, $|\ rnd_F(t\alpha) - t\alpha\ | \leq 0.5fmin_D$. Therefore, $t - rnd_F(t\alpha) \geq t(1-\alpha) - 0.5fmin_D$. Inequality $X > 0$ in the clause body implies that $t \geq fmin_D$. Thus, $t - rnd_F(t\alpha) \geq fmin_D(0.5 - \alpha)$. In order for the right hand side to be used as ε, it should be positive, that is $\alpha < 0.5$ should hold.
- $|\ t\alpha\ | \geq fmin_N$. Then, $e_F(t\alpha) = \lfloor \log_r |t\alpha| \rfloor + 1$ and

  $$|\ rnd_F(t\alpha) - t\alpha\ | \leq 0.5r^{\lfloor \log_r |t\alpha| \rfloor + 1 - p} \leq 0.5t\alpha r^{1-p}.$$

  Thus, $t - rnd_F(t\alpha) \geq t(1 - \alpha - 0.5\alpha r^{1-p})$. Condition (1) requires $r^{1-p} \leq 0.5$, i.e., $t - rnd_F(t\alpha) \geq t(1 - 1.25\alpha)$. Since $|\ \ t\alpha\ | \geq fmin_N$, $t > 0$ and $\alpha > 0$ also $t \geq fmin_N/\alpha$ should hold. Then, $t - rnd_F(t\alpha) \geq fmin_N(1 - 1.25\alpha)/\alpha$. To make the right-hand side expression positive $\alpha < 0.8$ should hold. Let $fmin_N(1 - 1.25\alpha)/\alpha$ be a candidate for ε.

Summarising the two cases we obtain that $\alpha < 0.5$ should hold and that ε can be chosen as $\min(fmin_D(0.5 - \alpha), fmin_N(1 - 1.25\alpha)/\alpha)$. Thus, for any t, $t - rnd_F(t\alpha) \geq \varepsilon$, the program is acceptable with respect to $\{p(1)\}$ and termination is proved.

It should be noted that in actual Prolog implementations termination of $p(t)$ with respect to the program above may also be observed if $\alpha = 0.5$. However, ISO standard for language independent arithmetic does not specify the behaviour of

$rnd_F(x)$ if x is exactly halfway between two values and rounding upwards in this case will result in non-termination.

Assume now that rnd_F has the *round toward zero* property. In this case $\mid rnd_F(x) \mid \leq \mid x \mid$ should hold for all x. Since $\alpha > 0$ and $t > 0$, $t\alpha > 0$ and $rnd_F(t\alpha) \geq rnd_F(0)$. Requirements of the standards also imply $rnd_F(0) = 0$. Thus, $rnd_F(t\alpha) \geq 0$, $rnd_F(t\alpha) \leq t\alpha$ and $t - rnd_F(t\alpha) \geq t(1-\alpha)$. As above, the body constraint $X > 0$ means $t \geq fmin_D$. Therefore, $t - rnd_F(t\alpha) \geq fmin_D(1-\alpha)$. The right hand-side is positive for all $0 < \alpha < 1$ and $\varepsilon = fmin_D(1-\alpha)$. Thus, if rounding happens toward zero, $p(t)$ terminates for all t for all α. □

Now consider another example, very similar to the one we have seen before, but significantly differing from it in its termination behaviour.

Example 7. Let P be the following program, extending P_3 from Example 1.

$$q(X) \leftarrow X > 0, X1 \ is \ X - \alpha, q(X1).$$

where $\alpha > 0$. In theory, $q(t)$ should terminate for all α and all t. Termination of $q(t)$ for $t \leq 0$ is obvious. Thus, we assume $t > 0$. After adorning the following program is obtained:

$$q^{\$1>0}(X) \leftarrow X > 0, X1 \ is \ X - \alpha, X1 > 0, q^{\$1>0}(X1).$$
$$q^{\$1>0}(X) \leftarrow X > 0, X1 \ is \ X - \alpha, X1 \leq 0, q^{\$1\leq 0}(X1).$$

Choosing $\mid q(t) \mid = t$, applying the same technique as above to infer ε and assuming rounding to nearest policy, one gets $rnd_F(t-\alpha) \leq (t-\alpha) + 0.5r^{1-p}(t-\alpha)$ if $t - \alpha \geq fmin_N$ and $rnd_F(t-\alpha) \leq (t-\alpha) + 0.5fmin_D$ if $t - \alpha < fmin_N$. Thus, in the first case $t - rnd_F(t-\alpha) \geq \alpha - 0.5r^{1-p}(t-\alpha)$. Since t is not bounded from above, one cannot claim the right hand side expression to be positive for all t. Therefore, we cannot find ε as above and suspect non-termination. Indeed, for t much bigger than α, $rnd_F(t-\alpha)$ might be computed to be equal to t, starting non-terminating execution. As we've seen in Example 1 SICStus Prolog [25] rounds $3 \times 10^{15} - 0.25$ to 3×10^{15}.

Precision of this analysis might be improved if information on the call set is available, such as that $Call(P, q(t_0)) \subseteq \{q(t) \mid t \leq t_0\}$ for a given goal $q(t_0)$. This would provide the desired upper bound on the values of t and we will be able to claim the right hand side expression to be positive for $\alpha > \frac{t_0 r^{1-p}}{2+r^{1-p}}$. This allows us to conclude termination of $q(t_0)$ for $\alpha > \max\left(\frac{t_0 r^{1-p}}{2+r^{1-p}}, 0.5fmin_D\right)$. The specified information about the call set can be deduced by the methods suggested in [19].

Similarly to the previous example, if rounding is done toward zero, termination can be shown for all α and all t by observing that $rnd_F(t-\alpha) \leq t - \alpha$ implies $t - rnd_F(t-\alpha) \geq \alpha$ and $\alpha > 0$ is given. □

Finally, consider a more realistic example.

Example 8. A bank has special conditions for deposits between 100 and 1000 euros on savings accounts. Every year it grants 5% and charges 1 euro for the account management. The conditions terminate when the deposit reaches 1000

euros. The following program computes the maximal number of years clients may enjoy the special conditions as a function of their investments.

$$
\begin{aligned}
&\mathit{years}(\mathit{Investment}, \mathit{Years}) \leftarrow \mathit{Investment} > 100,\\
&\quad \mathit{Investment} < 1000, \mathit{compute}(\mathit{Investment}, 1, \mathit{Years}).\\
&\mathit{compute}(\mathit{Balance}, Y, \mathit{Years}) \leftarrow \mathit{Balance} < 1000,\\
&\quad \mathit{NewBalance}\ \mathit{is}\ 1.05 * \mathit{Balance} - 1, Y1\ \mathit{is}\ Y + 1,\\
&\quad \mathit{compute}(\mathit{NewBalance}, Y1, \mathit{Years}).\\
&\mathit{compute}(\mathit{Balance}, Y, \mathit{Years}) \leftarrow \mathit{Balance} \geq 1000, \mathit{Years}\ \mathit{is}\ Y.
\end{aligned}
$$

We would like to prove termination of $\mathit{years}(f, v)$, where f is a floating-point number and v is a variable. Transforming the program as explained in [23], obtain:

$$
\begin{aligned}
&\mathit{years}(\mathit{Investment}, \mathit{Years}) \leftarrow \mathit{Investment} > 100,\\
&\quad \mathit{Investment} < 1000, \mathit{Investment} < 1000,\\
&\quad \mathit{compute}^{\$1<1000}(\mathit{Investment}, 1, \mathit{Years}).\\
&\mathit{compute}^{\$1<1000}(\mathit{Balance}, Y, \mathit{Years}) \leftarrow \mathit{Balance} < 1000,\\
&\quad \mathit{NewBalance}\ \mathit{is}\ 1.05 * \mathit{Balance} - 1, Y1\ \mathit{is}\ Y + 1,\\
&\quad \mathit{NewBalance} < 1000, \mathit{compute}^{\$1<1000}(\mathit{NewBalance}, Y1, \mathit{Years}).\\
&\mathit{compute}^{\$1<1000}(\mathit{Balance}, Y, \mathit{Years}) \leftarrow \mathit{Balance} < 1000,\\
&\quad \mathit{NewBalance}\ \mathit{is}\ 1.05 * \mathit{Balance} - 1, Y1\ \mathit{is}\ Y + 1,\\
&\quad \mathit{NewBalance} \geq 1000, \mathit{compute}^{\$1\geq 1000}(\mathit{NewBalance}, Y1, \mathit{Years}).\\
&\mathit{compute}^{\$1\geq 1000}(\mathit{Balance}, Y, \mathit{Years}) \leftarrow \mathit{Balance} \geq 1000, \mathit{Years}\ \mathit{is}\ Y.
\end{aligned}
$$

As explained above, in order to prove termination it is sufficient to show that there exists $\varepsilon > 0$, such that

$$
\begin{aligned}
&\mid\ \mathit{compute}^{\$1<1000}(\mathit{balance}, y, \mathit{years})\ \mid -\\
&\quad \mid\ \mathit{compute}^{\$1<1000}(\mathit{new\_balance}, y1, \mathit{years})\ \mid \geq \varepsilon,
\end{aligned}
$$

for all calls to $\mathit{compute}^{\$1<1000}$. The only inequality we have in the second clause is $\mathit{Balance} < 1000$, therefore $\mid \mathit{compute}^{\$1<1000}(\mathit{balance}, y, \mathit{years}) \mid$ is defined as $1000 - \mathit{balance}$ if $\mathit{balance} < 1000$ and as 0, otherwise. Then, in order to prove termination there should exists $\varepsilon > 0$, such that for all $100 < \mathit{balance} < 1000$

$$(1000 - \mathit{balance}) - (1000 - \mathit{rnd}_F(\mathit{rnd}_F(1.05\mathit{balance}) - 1)) \geq \varepsilon.$$

Assume that the rounding function has the *round to nearest* property. We are going to estimate $\mathit{rnd}_F(\mathit{rnd}_F(1.05\mathit{balance}) - 1)$. Since $\mathit{balance} > 100$, it holds that $1.05\mathit{balance} \geq \mathit{fmin}_N$ and $\mathit{rnd}_F(1.05\mathit{balance}) - 1 \geq \mathit{fmin}_N$. Thus, $\mathit{rnd}_F(\mathit{rnd}_F(1.05\mathit{balance}) - 1) \geq (1.05\mathit{balance}(1 - 0.5r^{1-p}) - 1)(1 - 0.5r^{1-p})$. Then, $\mathit{rnd}_F(\mathit{rnd}_F(1.05\mathit{balance}) - 1) - \mathit{balance} \geq \mathit{balance}(0.05 + 0.2625r^{2(1-p)} - 1.05r^{1-p}) - 1 + 0.5r^{1-p}$. In order the right-hand side expression to be used as ε,

it should be positive. First of all, (2) implies $0.2625r^{2(1-p)} - 1.05r^{1-p} + 0.05 > 0$. Second,

$$balance > \frac{1 - 0.5r^{1-p}}{0.05 + 0.2625r^{2(1-p)} - 1.05r^{1-p}} \tag{6}$$

should hold. One can show that the fraction on the right-hand side is monotonically increasing as a function of r^{1-p}. Condition (2) requires $r^{1-p} \leq 10^{-6}$, implying that if $balance > \frac{1-0.5*10^{-6}}{0.05+0.2625*10^{-12}-1.05*10^{-6}}$, then (6) holds for this *balance* and all possible values of r and p. Inequality $balance > 100$ implies the latter condition, and thus, (6). Therefore, $balance(0.05 + 0.2625r^{2(1-p)} - 1.05r^{1-p}) - 1 + 0.5r^{1-p} > 0$ and choosing ε as $balance(0.05 + 0.2625r^{2(1-p)} - 1.05r^{1-p}) - 1 + 0.5r^{1-p}$ proves termination. □

5 Toward the Automation of the Approach

In the previous section we have analysed a number of examples. The analysis followed the same pattern: infer the set of calls [19] (sometimes omitted in the examples, if the decrease can be shown for *all* values). We assume that the execution doesn't result in an exception as this obviously means termination. Next we apply the technique of [23] to infer level-mappings, construct acceptability decreases and try to prove them using the formulae recited in Section 3.

The most intriguing step of proving termination is the last one, namely, proving that there exists ε as required. Assume that the rounding is done to the nearest value. Then (5) implies that for all $x \in \mathcal{R}$ holds, that $lo(x) \leq rnd_F(x) \leq hi(x)$, where *lo* and *hi* are defined as

$$lo(x) = \begin{cases} x - 0.5fmin_D & \text{if } |x| < fmin_N \\ x - 0.5\,|x|\;\; r^{1-p} & \text{if } |x| \geq fmin_N \end{cases}$$

$$hi(x) = \begin{cases} x + 0.5fmin_D & \text{if } |x| < fmin_N \\ x + 0.5\,|x|\;\; r^{1-p} & \text{if } |x| \geq fmin_N \end{cases}$$

If the rounding is done toward zero, similar definitions for *lo* and *hi* are obtained from (3) and (4).

For the further analysis we need an underestimate for the size of the call to the head and an overestimate for size of the call to the recursive subgoal. These can be computed from the basic inequalities of the form $lo(x) \leq rnd_F(x) \leq hi(x)$ by using well-known principles of interval arithmetics, such as $a \leq x \leq b, c \leq y \leq d \Rightarrow a + c \leq x + y \leq b + d$, and repeatedly applying the following equivalences, seen as a term-rewriting system ($\rho \in \{>, \geq, \leq, <\}$):

$$lo(x)\ \rho\ y \Leftrightarrow \begin{cases} x - 0.5fmin_D\ \rho\ y & \text{if } |x| < fmin_N \\ x - 0.5\,|x|\;\; r^{1-p}\ \rho\ y & \text{if } |x| \geq fmin_N \end{cases}$$

$$hi(x)\ \rho\ y \Leftrightarrow \begin{cases} x + 0.5fmin_D\ \rho\ y & \text{if } |x| < fmin_N \\ x + 0.5\,|x|\;\; r^{1-p}\ \rho\ y & \text{if } |x| \geq fmin_N \end{cases}$$

Example 9. Recall Example 8. We need to underestimate the value of $rnd_F(rnd_F(1.05balance) - 1)$. Then, $lo(1.05balance) \leq rnd_F(1.05balance)$ and $lo(lo(1.05balance) - 1) \leq rnd_F(rnd_F(1.05balance) - 1)$. In order to determine the value of the underestimate we have to know whether $|lo(1.05balance) - 1| < fmin_N$ or $|lo(1.05balance) - 1| \geq fmin_N$. Applying the equivalences above results in a set of nine inequalities without either lo or $|\cdot|$. Together with the information obtained from (2) (`should_satisfy`) and from the program (`Balance > 100`) the latter inequalities are posed to a constraint solving system. The following queries[2] are the successful ones in ECL^{i}PS^e [3] (the variable `RP` corresponds to r^{1-p}):

```
(a) should_satisfy(RP,FMinN,FMinD),Balance>100,
    1.05*Balance >= FMinN, 1.05*Balance*(1-0.5*RP) >= FMinN + 1.
(b) should_satisfy(RP,FMinN,FMinD),Balance>100,
    1.05*Balance >= FMinN, 1.05*Balance*(1-0.5*RP) =< 1 - FMinN.
(c) should_satisfy(RP,FMinN,FMinD),Balance>100,
    1.05*Balance >= FMinN, 1.05*Balance*(1-0.5*RP) =< 1 + FMinN,
    1.05*Balance*(1-0.5*RP) >= 1 - FMinN.
```

Observe, that (b) and (c) succeed due to inability of ECL^{i}PS^e to solve non-linear constraints. This shortcoming can be resolved either by using more powerful solvers or by verifying, for example with Maple [27], that given `should_satisfy` and `Balance > 100`, the maximum of `1 - FMinN` is less than the minimum of `1.05*Balance*(1-0.5*RP)`, making, thus, `1.05*Balance*(1-0.5*RP) =< 1 - FMinN` impossible. Case (c) can be similarly eliminated. Thus, the only valid case is (a) and the minimum of $(1.05balance(1-0.5r^{1-p})-1)(1-0.5r^{1-p})-balance$n is to be found. □

At the next step we try to prove that there exists ε, as required by the acceptability condition. To do so, we search ε's for each clause and each pair—call to the head, call to the recursive subgoal—and take the smallest of them as shown in Example 6. Given a clause and a pair of calls, Maple can prove the existence of such an ε.

Example 10. If the minimum of $(1.05balance(1 - 0.5r^{1-p}) - 1)(1 - 0.5r^{1-p}) - balance$ is positive, it can be taken as ε. Otherwise, acceptability cannot be proved and non-termination is suspected. Denoting r^{1-p} by `RP` and running in Maple `minimize((1.05*Balance*(1 - 0.5*RP) - 1)*(1 - 0.5*RP) - Balance, Balance=100..infinity, RP=0..1.0e-6);` gives 3.9998955. This value is greater than 0 and, thus, can be used as ε, completing the termination proof. □

6 Conclusion

We have presented an approach to verification of termination for logic programs with computations depending on real numbers. To the best of our knowledge

[2] The syntax of ECL^{i}PS^e is slightly simplified.

this is the first work in this domain. Termination of numerical computations was studied by a number of authors [5,6,13,23].

In [5] it is claimed that an unchanged acceptability condition can be applied to programs in pure Prolog with arithmetic by defining the level mappings on ground atoms with the arithmetic relation to be zero. This approach ignores the actual computation, and thus, its applicability is restricted to programs using some arithmetic but not really relaying on them, such as *quicksort*. Dershowitz *et al.* [13] and Serebrenik and De Schreye [23] restricted their attention to integer computations only.

Apt *et al.* [6] provided a declarative semantics, so called Θ-semantics, for Prolog programs with first-order built-in predicates, including arithmetic operations. In this framework the property of strong termination, i.e., finiteness of all LD-trees for all possible goals, was completely characterised based on appropriately tuned notion of acceptability. However, this approach studied termination with respect to ideal real numbers and not with respect to actual floating point computations. Example 1 illustrates that these two notions of termination are orthogonal. We also believe that in order for a termination analyser to be of practical use termination with respect to floating point numbers should be considered.

More research has been done on termination analysis for constraint logic programs [9,20,22]. Since numerical computations in Prolog should be written in a way that allows a system to verify their satisfiability we can see numerical computations of Prolog as an *ideal constraint system*. Thus, all the results obtained for ideal constraints systems can be applied. Unfortunately, the research usually restricted to domains isomorphic to $\mathcal{N}$ [20], such as trees and terms. Ruggieri [22] provides theoretical characterisation of termination, but doesn't suggest a method to infer level-mappings required, nor does he study the influence of imprecision inherent to floating-point computations.

Much more attention to the floating point arithmetic was payed outside of the logic programming community [4,14,15,16,21,26]. However, the works we are aware of concentrated on precision of numerical computations. Some of them suggest alternative *exact* representations of real arithmetic, for example using continued fraction expansions [26], sequences of linear maps [14] or linear fractional transformations with non-negative integer coefficients [21]. Other authors apply abstract interpretation to infer whether the precision might be lost [4, 15]. These works tend, however, to ignore the use of denormalised numbers and the *to the nearest value* rounding policy, both being decisions usually made by system implementors. Moreover, to the best of our knowledge, there is no work done on relating the precision results obtained with the techniques above with termination analysis.

We also stress that even our results are couched in logic programming terminology, they are not limited to logic programming per se, as they focus on properties of floating point numbers and not on language aspects of logic programs.

References

1. ISO/IEC 10967-1:1994. *Information technology—Language independent arithmetic—Part 1: Integer and floating point arithmetic.* ISO/IEC, 1994.
2. ISO/IEC 13211-1:1995. *Information technology—Programming languages—Prolog—Part 1: General core.* ISO/IEC, 1995.
3. A. Aggoun, D. Chan, P. Dufresne, E. Falvey, H. Grant, W. Harvey, A. Herold, M. Geoffrey, M. Meier, D. Miller, S. Mudambi, S. Novello, B. Perez, E. van Rossum, J. Schimpf, K. Shen, P. A. Tsahageas, and D. H. de Villeneuve. *ECL^{i}PS^e User Manual. Release 5.3*, 2001.
4. Y. Aït Ameur, P. Cros, J. J. Falcón, and A. Gómez. An application of abstract interpretation to floating point arithmetic. In M. Billaud, P. Castéran, M.-M. Corsini, K. Musumbu, and A. Rauzy, editors, *Actes WSA'92 Workshop on Static Analysis*, volume 81–82 of *Bigre*, pages 205–212. Atelier Irisa, IRISA, Campus de Beaulieu, 1992.
5. K. R. Apt. *From Logic Programming to Prolog.* Prentice-Hall Int. Series in Computer Science. Prentice Hall, 1997.
6. K. R. Apt, E. Marchiori, and C. Palamidessi. A declarative approach for first-order built-in's in Prolog. *Applicable Algebra in Engineering, Communication and Computation*, 5(3/4):159–191, 1994.
7. M. Brady. Open Prolog. `http://www.cs.tcd.ie/open-prolog/`, 2001.
8. F. Bueno, D. Cabeza Gras, M. Carro, M. V. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog system. Reference manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. `http://www.clip.dia.fi.upm.es/`.
9. L. Colussi, E. Marchiori, and M. Marchiori. On termination of constraint logic programs. In U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming - CP'95,*, pages 431–448. Springer Verlag, 1995. LNCS 976.
10. D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analyzing the termination of definite logic programs with respect to call patterns. In ICOT Staff, editor, *Proc. of the Int. Conf. on Fifth Generation Computer Systems.*, pages 481–488. IOS Press, 1992.
11. S. Decorte and D. De Schreye. Termination analysis: some practical properties of the norm and level mapping space. In J. Jaffar, editor, *Proc. of the 1998 Joint Int. Conf. and Symp. on Logic Programming*, pages 235–249. MIT Press, June 1998.
12. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-based termination analysis of logic programs. *ACM TOPLAS*, 21(6):1137–1195, November 1999.
13. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Appl. Alg. in Eng., Comm. and Computing*, 12(1-2):117–156, 2001.
14. M. H. Escardó. PCF extended with real numbers. *Theoretical Computer Science*, 162(1):79–115, August 1996.
15. E. Goubault. Static analyses of the precision of floating-point operations. In P. Cousot, editor, *Static Analysis, 8th International Symposium*, volume 2126 of *LNCS*, pages 234–259. Springer Verlag, 2001.
16. E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: A simple abstract interpreter. In D. Le Métayer, editor, *Programming Languages and Systems, 11th European Symposium on Programming*, volume 2305 of *LNCS*, pages 209–212. Springer Verlag, 2002.

17. IEEE Standards Committee 754. *IEEE Standard for binary floating-point arithmetic, ANSI/IEEE Standard 754-1985*. Institute of Electrical and Electronics Engineers, New York, 1985. Reprinted in SIGPLAN Notices, 22(2):9-25, 1987.
18. IT Masters. MasterProLog Programming Environment. `http://www.itmasters.com/`, 2000.
19. G. Janssens, M. Bruynooghe, and V. Englebert. Abstracting numerical values in CLP(H, N). In M. V. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94*, pages 400–414. Springer Verlag, 1994. volume 844 of *LNCS*.
20. F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. In M. Maher, editor, *Proc. JICSLP'96*, pages 7–21. The MIT Press, 1996.
21. P. J. Potts, A. Edalat, and M. H. Escardó. Semantics of exact real arithmetic. In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science*, pages 248–257. IEEE Computer Society Press, 1997.
22. S. Ruggieri. *Verification and validation of logic programs*. PhD thesis, Universitá di Pisa, 1999.
23. A. Serebrenik and D. De Schreye. Inference of termination conditions for numerical loops in Prolog. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, Proceedings*, volume 2250 of *LNCS*, pages 654–668. Springer Verlag, 2001.
24. A. Serebrenik and D. De Schreye. Non-transformational termination analysis of logic programs, based on general term-orderings. In K.-K. Lau, editor, *LOPSTR 2001, 10th Int. Workshop, Selected Papers*, volume 2042 of *LNCS*, pages 69–85. Springer Verlag, 2001.
25. Swedish Institute of Computer Science. *SICStus User Manual. Version 3.9*. 2002.
26. J. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8):1087–1105, August 1990.
27. Inc. Waterloo Maple. Maple 7, 2001. `http://www.maplesoft.com/`.
28. W. Winsborough. Multiple specialization using minimal-function graph semantics. *Journal of Logic Programming*, 13(2/3):259–290, 1992.

Modular Control Flow Analysis for Libraries

Christian W. Probst*

FR. 6.2, Universität des Saarlandes, 66111 Saarbrücken, Germany
probst@cs.uni-sb.de

Abstract. One problem in analyzing object oriented languages is that the exact control flow graph is not known statically due to dynamic dispatching. However, this is needed in order to apply the large class of known interprocedural analysis. Control Flow Analysis in the object oriented setting aims at determining run-time types of variables, thus allowing to possibly targeted method implementations. We present a flow sensitive analysis that allows separate handling of libraries and thereby efficient analysis of whole programs.

1 Introduction

A typical problem with object oriented languages is the construction of the interprocedural control flow graph, which is usually needed for data flow analysis. It is complicated by the concept of *dynamic dispatch* where objects are sent messages at run-time. Depending on their actual run-time type those objects decide which implementation of the method should be called. That is, in order to statically determine which implementation of `test` will be called by `o.test()` we would need to know the possible run-time types of object `o`.

Static analysis is a common tool in compilers to help understanding what a program does. Many analyzers require the whole program to be available, i. e. also the code of all libraries probably used by the program. In the object oriented setting those libraries are heavily used. While the inspection of library code often can be arranged, e.g. by analyzing byte code instead of Java, the need to analyze the whole program results in re-analyzing large amounts of library code again and again. The JOSES project ([9],[15]) developed a Java compiler for embedded systems based on the CoSy system [2]. Here it turned out that the famous `HelloWorld`, while being a treat when analysis imperative languages, really imposes a big hurdle for Java compilers as one has to analyze and compile huge amounts of the Java libraries.

We are going to present a flow-sensitive technique for call graph construction based on control flow analysis. The solution to our analysis will be computed based on constraint graphs. This allows us to generate the constraint graphs for library methods in advance (and only once). During analysis we select the graphs for the methods to be analyzed and combine them with the graph of the user program. This graph is then solved.

* This work has in part been supported by the Esprit LTR project #28198 "JAVA and CoSy Technology for Embedded Systems"

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 165–179, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Our contribution is a scaling, flow-sensitive method to compute interprocedural control flow graphs. The analysis is flow-sensitive in local variables and arguments. Class and instance members are abstracted by a single variable for each member. This has the advantage that we do not need to further look into aliasing. However, it introduces dependencies between otherwise unrelated parts of the program. Nevertheless this allows us to even analyze multi-threaded programs.

Up to now only flow-insensitive methods are widely used (c. f. section 5). The proposed algorithm works on constraint graphs. For libraries, those graphs can be computed in advance, what can easily be combined with the compilation of those libraries. Furthermore, we introduce a compression step, which removes many superfluous edges and nodes from the generated graphs.

Of course there are some limitations that are mainly imposed by the framework being used in the JOSES project. Due to time constraints in the project there is no support for exceptions. Additionally, we perform class initialization for all classes at the beginning of program execution. A more general, analysis intrinsic problem is the handling of native methods. However, as long as one is able to obtain the *semantics* of the method one can model its effect by generating the abstraction file by hand.

We will first give a brief sketch of control flow analysis. After that we will present a simple example, give the rules of how to generate the constraints and show how to solve them. Based on this we will present our approach to modular analysis of libraries. The example methods will use a simplified set of Java statements that is given in figure 1.

$$
\begin{array}{rcl}
\mathit{method\_decl} & \rightarrow & \mathit{returntype}\ \mathit{methodname}\ (p_1, \cdots, p_n)\{\mathit{var\_decls}; \mathit{stmts}\} \\
\mathit{returntype} & \rightarrow & \mathit{classtype} \\
 & | & \mathit{void} \\
\mathit{stmts} & \rightarrow & \mathit{stmt}; \mathit{stmts} \\
 & | & \epsilon \\
\mathit{stmt} & \rightarrow & \mathit{assign} \\
 & | & \mathit{return} \\
 & | & \mathit{call} \\
\mathit{assign} & \rightarrow & \mathit{var}\texttt{=}\mathit{rhs} \\
\mathit{return} & \rightarrow & \texttt{return}\ \mathit{variable} \\
\mathit{call} & \rightarrow & \mathit{variable}\texttt{.m}(a_1, \cdots, a_n) \\
\mathit{var} & \rightarrow & \mathit{variable} \\
 & | & \mathit{member} \\
\mathit{member} & \rightarrow & \mathit{classtype.member} \\
 & | & \mathit{variable.member} \\
\mathit{rhs} & \rightarrow & \mathit{var} \\
 & | & \texttt{new}\ \mathit{ctype}() \\
 & | & \mathit{call} \\
a_i & \rightarrow & \mathit{variable}
\end{array}
$$

Fig. 1. Statements in the example language

For simplicity we have concentrated on method declarations and statements and left out the definition of classes. Those are assumed to be given, i. e. each method of course belongs to a class and there is the class hierarchy graph [5], which represents the inheritance relationship between classes. Without loss of generality we assume that all values are assigned to local variables before being used. Figure 2 shows the example program, consisting of several definitions of the method `set` in different classes. As can be seen, we explicitly state all uses of `this`.

```
class A {
  A a;
  void set (A this^1,A x^2) {}
}
class B extends A {
}
class C extends A {
  void set (C this^3,A x^4) {}
}
```

```
class test {
  A gen (test this^5) {
    A t^6=(new B())^7;
    return t^8;
  }^9
  void main (test this^10) {
    A va^11 = (new A())^12;
    C vb;
    va^13 = (this^14.gen(this^15))^16;
    vb^17 = (new C())^18;
    (va^19.set(va^20, vb^21))^22;
  }
}
```

Fig. 2. Example program

2 Control Flow Analysis

Control flow analysis (CFA) for functional languages computes for each subexpression the functions that it may evaluate to. Thereby it is possible to determine where the flow of control may be transferred to at that point. This means that control flow analysis in its usual area of analysing functional languages is *functional* in the sense that it takes functions as abstract values and traces them through the program being analysed. The analysis itself is similar to the computation of *definition–use–chains (DU)* known for imperative languages – both trace how definition points reach points of use [12]. While DU traces values assigned to variables, CFA does so for creations of function abstractions. The equivalent of the use of variables in DU is the point of a function application in CFA. For object oriented languages the CFA traces sets of either classes or objects a variable may point to at run-time. The point of usage, however, is still a function application or, in object oriented technology, a method send.

Different kinds of CFA may be distinguished based on the amount k of context they take into account [12,14]. This can be compared to the call-string approach used in traditional data flow analysis and allows the analysis to remember at most the last k dynamic call points. Our approach has been based

on a 0–CFA in order to keep complexity low but it can easily be extended to $k > 0$. However, the call string approach does not seem to be the optimal approach to analysing object oriented programs. Therefore, we are planning to adopt techniques proposed by Agesen [1].

`rtype` $m(p_1^{l_1}, \cdots, p_n^{l_n})\{\cdots\}^{l_r}$	*method definition*
var^{l_1} `=` rhs^{l_2}	*assignment*
$variable^{l_1}$.*member*	*class or instance member*
`return` $variable^{l}$	*return*
$variable^{l}$.`m`$(a_1^{l_1}, \cdots, a_n^{l_n})$	*call*

Fig. 3. How to label (parts of) statements

3 The Analysis

Now we are ready to present our analysis. It consists of five steps:

- assign labels to interesting program points
- compute use-definition chains
- generate constraints
- generate and optimize the constraint graph
- compute the strongly connected components on the graph
- solve the constraints

When generating constraint graphs for libraries the final step is skipped as the graph is stored to a file.

3.1 Assigning Labels

Assigning labels is quite straightforward. We simply label each part of the program that may return an object or result in an object with a label $l \in \mathbf{N}$. Figure 3 shows all possible statements and their labels. Note that labels propagate along derivations in the grammar from figure 1, e.g. variable arguments receive their label from the associated a_i. Furthermore, $l_i \neq l_j$ if $i \neq j$.

Having labeled the program points, for each non defining use of a label we compute the set of labels of definitions that may reach it. As usual, a defining use of a label is on the right hand side of an assignment. The intra-procedural analysis is straightforward. Its result is stored in a function $UD : \mathbf{N} \rightarrow \wp(\mathbf{N})$.

3.2 Generate Constraints

The control flow analysis will compute sets of classes for labeled program points. These are stored by means of the function $C : \mathbf{N} \rightarrow \wp(\mathbf{Classes})$. For each statement in the program being analyzed we need to generate a set of constraints that model the flow of values through the program. This generation is performed by functions $C_\star : \mathbf{Stmt} \rightarrow \wp(\mathbf{Constraints})$ for statements and method declarations (c. f. figure 4) and $C^R_\star : \mathbf{Rhs} \times \mathbf{N} \rightarrow \wp(\mathbf{Constraints})$ and $C^L_\star : \mathbf{Var} \times (\mathbf{N} \cup \{\epsilon\}) \rightarrow \wp(\mathbf{Constraints})$ for using and defining occurrences of expressions (c. f. figure 5).

$$
\begin{array}{ll}
[method\ definition] & C_\star[rtype\ m(p_1^{l_1}, \cdots, p_n^{l_n})\{body\}^{l_r}] := C_\star[body] \\
 & \quad labels(m) := (l_r, [l_1, \cdots, l_n]) \\
[assignment] & C_\star[var^{l_1} = rhs^{l_2}] := C^L_\star[var, l_1] \cup C^R_\star[rhs, l_2] \cup \{C(l_2) \subseteq C(l_1)\} \\
[return] & C_\star[return\ variable^l] := C^R_\star[variable, l] \cup \{C(l) \subseteq C(l_r)\} \\
 & \qquad \text{where } l_r \text{ is the label of the method body analyzed} \\
[method\ send] & C_\star[variable^l.m(var_1^{l_1}, \cdots, var_n^{l_n})] := \\
 & \qquad C^R_\star[variable^l.m(var_1^{l_1}, \cdots, var_n^{l_n}), \epsilon]
\end{array}
$$

Fig. 4. Generating constraints for statements

Figure 4 gives the rules how to generate constraints for statements. For [*method definition*] we store the labels of the method's parameters and its body by means of the function $labels : \mathbf{Methods} \rightarrow \mathbf{N} \times \mathbf{N} \times \mathbf{N}^n$. When analyzing a call we will use this information to let actual arguments flow into the body and the method's return value flow back to the call. For [*assignment*] we have to generate constraints for the left and the right hand side and have to assure that the values from the right hand side flow to the left ($C(l_2) \subseteq C(l_1)$). For the [*return*] statement we need to generate constraints for the returned variable and then let it flow to the label of the method's body. When handling calls, l_r will be connected to the label of the variable where the return value will be stored.

Figure 5 gives the functions that generate constraints for expressions. For a defining occurrence of a local [*variable*] we do not need to do anything. The variable's value is stored in its label and this is assured by the constraints we have generated for the assignment. For a using occurrence labeled l in statement s we will collect and unify all possible definitions that might reach this use. However, we have precomputed this as $UD(l)$, which gives us the set of all labels that may define the variable labeled l.

Things are not that easy with [*instance member*] and [*class member*]. As these are shared between all objects of a class or all methods of a class instance, there may be flow between members occurring in different methods. To capture this we introduce two functors `LOAD` and `STORE` that will be handled by the

$$
\begin{array}{ll}
[localvariable] & C^R_\star[variable, l] := \bigcup_{l' \in UD(l)} \{C(l') \subseteq C(l)\} \\
 & C^L_\star[variable, l] := \{\} \\
[instance\ member] & C^R_\star[variable^{l_m}.member, l] := \\
 & \qquad \{\texttt{LOAD}_{\ member}(l_m, l)\} \cup C^R_\star[variable, l_m] \\
 & C^L_\star[variable^{l_m}.member, l] := \\
 & \qquad \{\texttt{STORE}_{\ member}(l_m, l)\} \cup C^R_\star[variable, l_m] \\
[class\ member] & C^R_\star[classtype^{l_m}.member, l] := \\
 & \qquad \{\texttt{LOAD}_{\ member}(l_m, l)\} \cup \{\{classtype\} \subseteq C(l_m)\} \\
 & C^L_\star[classtype^{l_m}.member, l] := \\
 & \qquad \{\texttt{STORE}_{\ member}(l_m, l)\} \cup \{\{classtype\} \subseteq C(l_m)\} \\
[method\ send] & C^R_\star[variable^l.m(var_1^{l_1}, \cdots, var_n^{l_n}), l_t] := C^R_\star[variable, l] \cup \\
 & \qquad \bigcup_{i=1\cdots n} C^R_\star[var_i, l_i] \cup \{\texttt{CALL}_{\ m}(l, l_t, [l_1, \cdots, l_n]\} \\
[creation] & C^R_\star[new\ C(), l] := \{\{C\} \subseteq C(l)\}
\end{array}
$$

Fig. 5. Generating constraints for expressions

$labels(test\_gen)$	$= (9, [5])$	$\{B\} \subseteq C(7)$	$C(7) \subseteq C(6)$
		$C(6) \subseteq C(8)$	$C(8) \subseteq C(9)$
$labels(test\_main)$	$= (\epsilon, [10])$	$\{A\} \subseteq C(12)$	$C(12) \subseteq C(11)$
		$C(10) \subseteq C(14)$	$C(10) \subseteq C(15)$
		$\texttt{CALL}_{\ gen}(14, 16, [15])$	$C(16) \subseteq C(13)$
		$\{C\} \subseteq C(18)$	$C(18) \subseteq C(17)$
		$C(13) \subseteq C(19)$	$C(13) \subseteq C(20)$
		$C(17) \subseteq C(21)$	$\texttt{CALL}_{\ set}(19, \epsilon, [20, 21])$
$labels(A\_set)$	$= (\epsilon, [1, 2])$		
$labels(C\_set)$	$= (\epsilon, [3, 4])$		

Fig. 6. Constraints and *labels* for the example program

constraint solver. Both have two arguments: the label of the variable contains the object, whose member is loaded/stored, and the label where this member's data shall flow from/to.

Another expression is the generation of objects. As these [*new*]s generate the information we want to propagate they are modeled by filling the $C(l)$s. The singleton set with the class name is added to the set associated with the expression's label.

Last but not least we model the flow in the case of method sends. First we need to know the possible types of the object, which the method is sent to, and the arguments to the call. As all these are using expressions we just generate

```
let G = (N, E) be the constraint graph generated
worklist = [n ∈ N, #predecessors(n) == 1, n not special];
while (worklist ≠ []){
    n = head(worklist); worklist = tail(worklist);
    n_p = predecessor(n);
    succs = successors(n);
    l = label(n);
    delete n;
    ∀n_s ∈ succs{
        add edge (n_p, n_s);
        if (n_s is functor node)
            replace occurrences of l by label(n_p)
        if (predecessors(n_s) == 1)
            worklist = append(worklist, n_s);
    }
}
delete all n ∈ N where n is no special node
    and #predecessors(n) == #successors(n) == 0
```

Fig. 7. Optimizing constraint graphs

constraints with $C_\star^R$. Having done so, we forward the computed values to all methods that might be called at this point.

The usual approach is to add *conditional* constraints for each possible target method. However, this may add a lot of constraints that need to be checked each time the call is evaluated by the solver. A prominent example in the Java world is `toString` that is implemented in many classes. When called on an object of type `Object` the resulting constraint system contains a conditional constraint for each definition of `toString`. To avoid this, we introduce a third functor `CALL` that again will be handled by the constraint solver. As its arguments it gets the label of the object which m is sent to, the label where to store the method's result and the labels of the arguments. Together with the information stored in *labels* and the class hierarchy graph [5], the solver will be able to add the necessary edges to model the flow of information in the analyzed program.

Figure 6 shows the constraints that are generated for our example program, that is the constraints in $C_\star$.

3.3 Generate and Optimize the Constraint Graph

In order to efficiently solve the constraints we first transform them. A common representation for efficient solutions is a directed graph where, in our case, the nodes represent the $C(l)$ and functor occurrences. The edges represent the dependencies between labeled program points. The graph is constructed as follows:

- for each constraint $C(l_1) \subseteq C(l_2)$ insert nodes n_{l_1}, n_{l_2} and a directed edge (n_{l_1}, n_{l_2})

- for each `LOAD` $_m(l_o, l)$ functor add nodes $n_L = n_{LOAD_m(l_o,l)}$, n_l, n_{l_o} and an edge (n_{l_o}, n_L)
- for each `STORE` $_m(l_o, l)$ functor add nodes $n_S = n_{STORE_m(l_o,l)}$, n_l, n_{l_o} and an edge (n_{l_o}, n_S)
- for each `CALL` $_m(l_o, l_t, [l_1, \cdots, l_n])$ add nodes $n_C = n_{CALL_m(l_o,l_t,[l_1,\cdots,l_n])}$, n_{l_o}, n_{l_t}, n_{l_i} and edge (n_{l_o}, n_C)
- for each constraint $\{C\} \subseteq C(l)$ initialize $D(l)$ with $\{C\}$

$D : N \to \wp(\mathbf{Classes})$ contains the sets of classes for each labeled program point. In addition there is a function $\rho : \mathbf{Classes} \times \mathbf{Members} \to \wp(\mathbf{Classes})$ that stores information for class and instance members. Furthermore, we have a function $label : N \to \mathbf{N}$ that returns the label l represented by a node n.

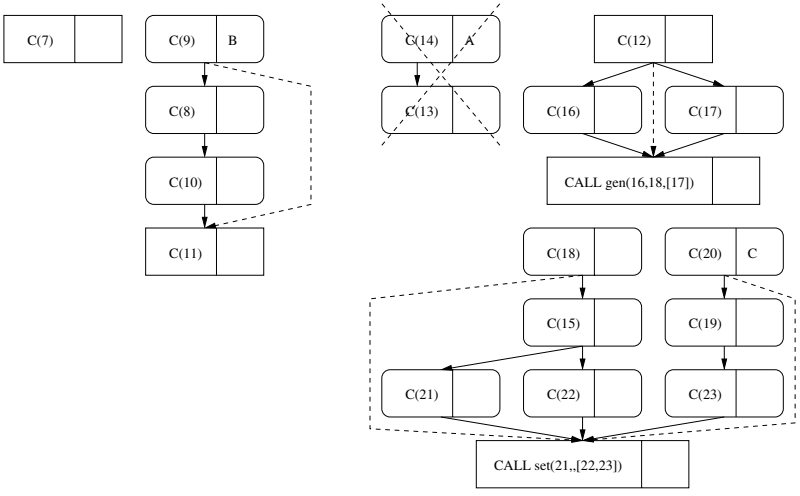


Fig. 8. Generated (solid edges) and optimized (dashed edges) graph

Table 1. Node and edge counts for examples from the Java libraries

Class	bytecode size	Nodes	Edges	opt. Nodes	opt. Edges
`java.lang.Object`	1514	123	110	45	15
`java.lang.System`	3436	1221	1075	424	163
`java.lang.Integer`	7910	735	673	252	97

Unfortunately, the generated graph will be rather big. E. g. the part for method `gen` consists of nodes n_7, n_8, n_9, n_{10} and n_{11} (c. f . figure 8). However, all that is done is to forget about the argument in n_7 and forward the set

$\{B\}$ from n_8 to n_{11}. What we will do is to optimize the graphs before solving them. This step is done by the algorithm *optimize* given in figure 7. It will delete those chains that just propagate any input over a chain of edges through the constraint graph. However, it will leave *special* nodes untouched. Those nodes are the special functor nodes `LOAD` , `STORE` and `CALL` and the method's entry/exit nodes. This approach can be compared to projection merging from [18]. Figure 8 shows the graph generated for the example program. The optimized graph is given by the dashed lines, i. e. the nodes in between source and target of dashed edges and the crossed out nodes have been deleted. Table 1 shows some examples for graph sizes generated for library classes before and after optimization. The total size of the graph for *HelloWorld.java* is 25272 nodes and 22468 edges in the unoptimized graph and 10106 nodes and 4043 in its optimized sibling.

In a next step we compute the strongly connected components on the graph. As the nodes and edges resemble sets and inclusion constraints between those sets, we know that all nodes in such a component will end up with the same set of classes. This step results in a set of sets of nodes $SCC \subset \wp(N)$ with $\bigcup_{s \in SCC} s = N$ and a mapping $scc : N \to SCC$ that identifies each node with the strongly connected component it is in. We will see later that the special nodes `CALL` , `LOAD` and `STORE` share a special property - they will always be the single node in their strongly connected component as they may never have outgoing edges. This allows us to speak of *normal*, *call*, *load* and *store* components, depending on the nodes contained within. Let $G_{SCC} = (SCC, E_{SCC}, N)$ be the graph that results from G by replacing the nodes N by the strongly connected components they belong to. Only edges between different components are present in G_{SCC}. Obviously, the resulting graph is acyclic. As the solver will visit the nodes in reverse post order, the next step is to compute this order on the graph.

Just like nodes, strongly connected components are associated with the set of classes that may turn up at the nodes they represent. Similar to D we define $D' : SCC \to \wp(\mathbf{Classes})$. For $s \in SCC$ the initial value is defined as $D'(s) := \bigcup\{D(n) | n \in N, scc(n) = s\}$.

Having computed the graph for a procedure it will be stored in a directory hierarchy comparable to that of the Java classes. I. e. for each procedure we save the information about nodes, edges and strongly connected components to the file. At compile time we will check whether the precompiled information is still valid - if so, the graph will be read in and unified with the overall graph for the whole program.

The worklist is divided into four parts according to the types of nodes - the first group contains the `STORE` nodes, the second one the `LOAD` nodes. The heuristic is to first evaluate all stores that have been scheduled before loading a class or instance member. The third group consists of `CALL` nodes. These three groups share the property that they may cause changes to the component graph. The fourth and last group are the *normal* components. Every group internally is sorted according to the reverse post ordering of the components.

```
let   G_SCC = (SCC, E, N) be the optimized constraint graph
      with SCC ⊂ ℘(N) and scc : N → SCC as defined above.
      Called = {scc(n)| n is root node in an entry procedure of the program}
for all s ∈ SCC with #predecessors(s) == 0, D'(s) /=∅,
            proc(s) ∈ Called: addtoworklist(s);
while (worklist /= []){
      s = head(worklist);worklist = tail(worklist);
      D_s = ⋃{D'(p)|(p, s) ∈ E}
      if D_s = D'(s)
          continue;
      ∀(s, n) ∈ E : addtoworklist(n)
      switch s {
          normal:
              break;
          LOAD m(l_o, l):
              ∀c ∈ D'(n_{l_o})
                  addedge(scc(membernode(c, m)),scc(n_l),G);
              break;
          STORE m(l_o, l):
              ∀c ∈ D'(n_{l_o})
                  addedge(scc(n_l),scc(membernode(c, m)),G);
              break;
          CALL m(l_o, l_t, [l_1, ··· , l_n]):
              targets = lookup(D'(n_{l_o}), m);
              ∀p ∈ targets where labels(p) = (l_r, [p_1, ··· , p_n]){
                  addedge(scc(n_{l_i}),scc(n_{p_i})) and (scc(n_{l_r}),scc(n_{l_t}),G);
                  Called = Called ∪ {p};
              }
              break;
      }
}

function add(a,b) { a = a ∪ b;}

function addedge(p,s,G) {
      if (p, s) ∈ E_G return;
      add edge (p, s) to G;
      if RPO(p) > RPO(s) {
          if there is a path from s to p
              create a new scc;
          recompute RPO for the tree fragment that contains p and s;
      }
      addtoworklist(s);
}
```

Fig. 9. Solving Constraint Graphs

3.4 Solving the Graph

Having constructed the optimized constraint graph, we apply the algorithm given in figure 9. It is worklist based and will propagate already computed values of D' along the edges. As described below some magic occurs whenever a functor node is hit.

The `LOAD` $_{member}(l_o, l)$ functor is generated for using occurrences of class or instance members, i. e. expressions of the form $(o^{l_o}.member)^l$. The solver uses $D'(scc(n_{l_o}))$ to look up the types of objects that may be stored in o. The set of types of objects that may occur in $member$ then is $\bigcup\{\rho(c, member)|c \in D'(scc(n_{l_o}))\}$. These must be stored in $D'(scc(n_l))$. To establish this, we add edges from the components representing $c.member$ to $scc(n_l)$ and add $scc(n_l)$ to the worklist.

The handling of the `STORE` $_{member}(l_o, l)$ functor is quite similar. Again, we need to look up the possible types of objects in $D'(scc(n_{l_o}))$. For each class c from this set we add an edge from $scc(n_l)$ to the component representing $\rho(c, member)$. By adding those components to the worklist, we ensure that the set $D'(scc(n_l))$ flows to them. If their value has been changed on any of its arguments, we need to reanalyze all those `LOAD` nodes in any procedure, that may load one of those values. This is easily done as for every `LOAD` already visited there is an edge outgoing from the $\rho(c, member)$ component. This ensures that only those `LOAD` nodes are added to the worklist, that belong to procedures that have already been analyzed, i. e. might be called at run-time.

The `CALL` $_m(l_o, l_t, [l_1, \cdots, l_n])$ functor needs to mimic the behavior of dynamic dispatching. It does so with the help of the class hierarchy graph [5]. Resulting from e. g. $(o^{l_o}.m(a_1^{l_1}, \cdots, a_n^{l_n}))^{l_t}$, when handling this functor the solver must first lookup the possible run-time types of o in $D'(scc(n_{l_o}))$. From the class hierarchy graph it then gets the possibly called implementations of m. For each m it gets the label of its body and its arguments from $labels(m) = (l_r, [l_{p_1}, \cdots, l_{p_n}])$. Now it can add edges between the calls actual arguments and the parameters and between $scc(node(l_r))$ and the target label $scc(node(l_t))$. In addition to adding edges we may have to collapse strongly connected components into a single one whenever the added edge is a backward one.

For the example program, the solved graph is given in figure 10. Based on the run-time type of $C(12)$ edges to the arguments of A_set two edges have been added based on the information from $labels(test\_gen) = (11, [7])$. Furthermore, as va's run-time type has been computed to be B have been added. This method has been excluded to keep the example simple.

4 Results

We have implemented the analysis in `JoC`, the `JOSES` compiler [15]. We will first give an overview of analysis-times for some example programs. The first program is `HelloWorld.java`. As we have learned in the JOSES project, being able to compile and analyze this program means to be able to analyze almost the whole Java framework. The second program is a simple thread test program and the

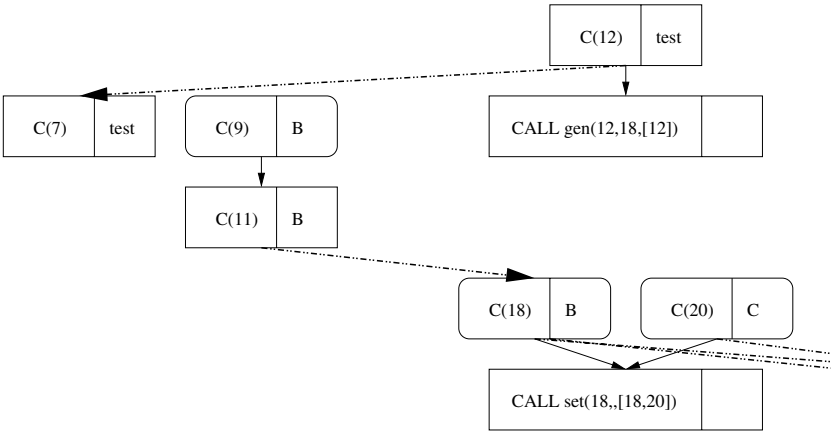


Fig. 10. The solved constraint graph

third one is `compress.java`. The measurements have been performed on a 750 MHz Pentium `III` with 384 MB memory.

Table 2 gives an overview of several numbers - for each program we have run the compiler without optimizations, with rapid type analysis and with rapid control flow analysis. Additionally we give the analysis-time for rapid control flow analysis when using the precompiled libraries.

Table 2. Analysis Times and Graph Sizes

	analysis time	#nodes	#edges	#procedures	compile time
`HelloWorld.java`					
normal	-	-	-	1736	213.1
RTA	5.6	-	-	379	100.1
RCFA	8.9	10062	4026	269	95.4
precompiled	4.4	10062	4026	269	91.2
`threads.java`					
normal	-	-	-	1931	290.9
RTA	7.8	-	-	525	155.3
RCFA	11.4	12096	4861	345	136.5
precompiled	5.9	12096	4861	345	130.6
`compress.java`					
normal	-	-	-	2266	413.7
RTA	12.2	-	-	542	205.0
RCFA	18.5	14784	6053	414	196.1
precompiled	10.9	14784	6053	414	188.6

There are several interesting things to note. The first is that even with an analysis taking more time then another, the total compile-time decreases. This is due to the increased number of procedures that are deleted by the more sophisticated analysis. The other thing is that rapid control flow analysis with precompiled libraries is even faster then rapid type analysis. However, the time needed for `RTA` is mainly spent in extracting data from the program. Thus, `RTA` would also benefit from a precompilation step, e. g. a file that contains for every procedure the set of classes it creates and the call statements in the procedure.

One of our claims was that the increased accuracy of the interprocedural control flow graph will speed up e. g. data flow analysis and increase the accuracy of the analysis' result. As an example for the first point we have implemented a Null Pointer Check-Analysis `NPC`. Such a check must be performed before a member of an instance is accessed to verify that the instance is not `null`. However, often the same object is used to repeatedly call methods on it or to access members of the object. The interprocedural analysis maps objects to whether the object has been checked already or not. The analysis has been implemented using the Program Analyzer Generator [11]. It uses the control flow graph generated by either RTA or RCFA as a basis for propagation. Table 3 gives several kinds of values - first we give counts for static calls, dynamically dispatched calls and resolved calls. The fifth column gives the run-time for the PAG analysis.

Table 3. Resolved Calls and NPC analysis-time

	#calls			NPC
	static	dispatched	resolved	analysis [s]
`HelloWorld.java`				
RTA	5978	18	5923	10.0
RCFA	5978	10	5931	1.0
`threads.java`				
RTA	6269	35	6588	14.3
RCFA	6269	22	6601	5.4
`compress.java`				
RTA	6477	108	6818	17.1
RCFA	6477	74	6852	8.1

5 Related Work

Recently, there has been a lot of work in the area of analyzing incomplete programs [4,8,16,17]. However, most of those analyzes combine with some kind of points to analysis. We are mainly investigating scalability of control flow analysis. That's why we did not add any additional functionality in order to be able to identify reasons for success or failure more clearly.

On the other hand, authors and compiler writers emphasize flow and context insensitive techniques. In the area of call graph construction and elimination of dynamic method calls, rapid type analysis [3] and extended versions [20] are the most widely used techniques. However, their precision suffers from the insensitivity. As a result, many quite obvious optimizations cannot be applied. E. g. rapid type analysis would not be able to detect that the call to `set` in the example program will never go to the implementation in class C. Admittedly, however, rapid type analysis is faster than the flow sensitive approaches. If we look at our results this seems to be neglectable compared to the total compile time.

6 Conclusions and Further Work

We have presented an algorithm to perform scalable control flow analysis for object oriented languages. Due to the large amount of library code used in those languages, it is worth to apply as much precompilation to those libraries as possible. By translating them into optimized constraint graphs and applying a sophisticated solving algorithm, we have been able to get scalable results.

Our first target is to continue evaluating the analysis on more real-world programs and to check the effect on other interprocedural data flow analyzes. Furthermore, the abstraction applied to run-time types is still rather unsatisfying. What we are going to look at in future research is how far we can refine the abstraction used without losing scalability. Additionally we will add enhanced context sensitivity by adapting techniques from [1].

I would like to thank Jörg Bauer, Marc Langenbach and Reinhard Wilhelm for insightful discussions and for proof reading preliminary versions of this paper and the members of the JOSES project for the technical support. The analysis described has been implemented in the `CoSy` compiler construction framework [2].

References

1. O. Agesen. The Cartesian Product Algorithm: Simple and Precise Typing of Parametric Polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 2–26, 1995.
2. M. Alt, U. Aßmann and H. van Someren. Cosy Compiler Phase Embedding with the CoSy Compiler Model. In *Proceedings of the Conference on Compiler Construction*, 1994.
3. D. F. Bacon. Fast and Effective Optimization of Statically Typed Object-Oriented Languages. Technical Report *CSD-98-1017*, University of California, Berkeley, Oct. 1998.
4. R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Symposium on Principles of Programming Languages*, pages 133–146, 1999.
5. J. Dean, D. Grove and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. Technical Report *01-12-94*, Washington University, Dec. 1994.
6. H. Dewes and C. W. Probst. Static Method Call in Java. In *Proceedings of the JOSES Workshop*, 2001.

7. M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Conference on Programming Language Design and Implementation*, pages 85–96, 1998.
8. C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Trans. Programming Languages and Systems*, 21(2), pages 370–416, Mar. 1999.
9. D. Genius, U. Aßmann, P. Fritzson, H. Sips, R. Kurver, R. Wilhelm, H. Schepers and T. Rindborg. Java and CoSy Technology for Embedded Systems: the JOSES Project. In *Proceedings of the Conference on Electronic Commerce, Multimedia, Embedded Systems and Technologies for Business Processes* Stockholm, 1999.
10. R. T. Hood and K. Kennedy. A Programming Environment for Fortran. In *Proceedings of the 18th Annual Hawaii International Conference on System Sciences*, pages 625–637, 1985.
11. F. Martin. Generation of Program Analyzers. Ph.D. Thesis, Universität des Saarlandes, 1999.
12. F. Nielson, H. R. Nielson and C. L. Hankin. Principles of Program Analysis. Springer-Verlag, 1999.
13. C. W. Probst. Flow Sensitive Call Graph Construction For Java. In *Proceedings of the JOSES Workshop*, 2001.
14. C. W. Probst. Control Flow Analysis – An Overview. The JOSES Consortium, 1999.
15. T. Ritzau, M. Beemster, F. Liekweg, C. W. Probst. JOC - The JOSES Compiler Presented at *The Embedded Systems Show*, London, 2000.
16. A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. Technical Report 423, Rutgers University, Oct. 2000.
17. A. Rountev and B. G. Ryder. Practical points-to analysis for programs built with libraries. Technical Report 410, Rutgers University, Feb. 2000.
18. Z. Su, M. Fähndrich and A. Aiken. Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs. In *Symposium on Principles of Programming Languages*, pages 81–95, 2000.
19. V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vall'ee-Rai, P. Lam, E. Gagnon and C. Godin. Practical virtual method call resolution for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application*, pages 264–280, 2000.
20. F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application*, pages 281–293, 2000.

An Efficient Inclusion-Based Points-To Analysis for Strictly-Typed Languages*

John Whaley and Monica S. Lam

Computer Systems Laboratory
Stanford University
Stanford, CA 94305
{jwhaley, lam}@stanford.edu

Abstract. We describe the design and implementation of an efficient inclusion-based points-to analysis for strictly-typed object-oriented languages. Our implementation easily scales to millions of lines of Java code, and it supports language features such as inheritance, object fields, exceptional control flow, type casting, dynamic dispatch, and reflection.
Our algorithm is based on Heintze and Tardieu's Andersen-style points-to analysis designed originally for C programs. We have improved the precision of their algorithm by tracking the fields of individual objects separately and by analyzing the local variables in a method in a flow-sensitive manner. Our algorithm represents the semantics of each procedure concisely using a sparse summary graph representation based on access paths; it iterates over this sparse representation until it reaches a fixed point solution. By utilizing the access path and field information present in the summary graphs, along with minimizing redundant operations and memory management overheads, we are able to quickly and effectively analyze very large programs. Our experimental results demonstrate that this technique can be used to compute precise static call graphs for very large Java programs.

1 Introduction

Points-to analysis is a fundamental technique whose results are useful for program optimizations, program comprehension tools, and software engineering aids. In particular, in the context of object-oriented programming languages like Java, points-to analysis is useful for *concrete type inferencing*. By knowing an object's concrete type statically, a compiler can replace dynamic dispatches with inlined methods to reduce the run-time overhead and enable further optimizations. Moreover, the availability of a more precise call graph is generally useful to any program comprehension tool.

Previous work on concrete type inferencing ranged from rather weak techniques such as Class Hierarchy Analysis (CHA)[3] and Rapid Type Analysis RTA[2] to more sophisticated algorithms that analyze the use of pointers in the

* This material is based upon work supported in part by the National Science Foundation under Grant No. 0086160 and an NSF Graduate Student Fellowship.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 180–195, 2002.
© Springer-Verlag Berlin Heidelberg 2002

program[6,7,9,11]. One of the major concerns in the design of such algorithms is the trade-off between precision and speed of the analysis.

Many of the pointer alias analyses developed in the past were designed for analyzing C programs. Previously, the only pointer alias analysis deemed scalable was Steensgaard's context-insensitive and flow-insensitive algorithm[8]. With a near-linear complexity, the technique is fast but imprecise. The algorithm is unification-based; as such, pointers are either unaliased or pointing to the same set of locations. While the pointer analysis proposed by Andersen is also flow-insensitive and context-insensitive, it is more precise because it is inclusion-based[1]. That is, two aliased pointers may point to overlapping but different sets of locations. However, with an $O(n^3)$ complexity, it was considered not scalable to real programs. Recently, Heintze and Tardieu proposed an optimization technique (CLA) that allows Andersen's technique to scale to programs with a million lines of code[5].

This paper adapts and extends the CLA algorithm to the problem of pointer analysis in Java. We have developed an equally scalable points-to analysis for Java programs so that application developers needing pointer information can count on having at least Andersen's level of precision without having to be concerned with the cost of the analysis. We show that there is no reason to resort to a less precise pointer analysis for concrete type inferencing.

We have extended the CLA algorithm to handle all the Java constructs, which include virtual method invocations, type casts, exceptions, unknown and escaping pointers. In addition, we improved the CLA algorithm in three ways:

1. **A field-sensitive pointer analysis.** Most C pointer alias analyses are *field-independent*. All the fields in a structure are modeled as having the same location; thus, a write into one field writes to all the fields in the structure. Heintze and Tardieu experimented with another analysis which they termed *field-based*. Each field in each structure type is represented by a variable. Any pointer stored into a field of a particular type of structure is modeled as storing into the variable representing the field of type. They showed some preliminary empirical results indicating that the field-based approach runs much faster than the field-independent approach. Notice that since C is not type-safe, a field-based approach may not generate a conservative answer for some C programs.

   There is yet another variant on how fields can be handled. We say that a points-to analysis is *field-sensitive* if it tracks the individual fields of individual pointers. That is, a store to a field via a pointer p only affects the objects pointed to by p and only in that field. A field-sensitive analysis is strictly more precise than either a field-based or a field-independent analysis. In Java, the more precise the points-to analysis results are, the smaller the call graph gets. In other words, a less precise analysis may produce a much larger call graph, which would end up making the algorithm much slower. Since Java is strictly typed, the field information can be fully exploited without compromising the soundness of the algorithm. Our algorithm is field-sensitive, and

our experimental results show that this added precision greatly reduces the running time of the algorithm.

2. **Flow sensitivity within a method.** Our algorithm analyzes local pointers in a method flow-sensitively. This is especially important for Java because, aside from the primitive types, all local variables in Java are pointers to heap objects. At the bytecode level, the same stack locations are used to point to different objects within the same procedure.

   The first step of our two-step algorithm converts the bytecode for each method into a sparse representation which we call a *summary graph*. A summary graph captures the semantics of the operations in a method; it records the calls made but not the semantics of the methods that are called. A summary graph contains:
   - the data accessed, expressed in terms of access paths based off the parameters and global variables,
   - the weak updates made in the procedure, and
   - an outgoing parameter map showing all the calls made and the parameters passed at each call site.

   We use a flow-sensitive data-flow algorithm to generate this representation so as to model the updates on local variables precisely. The interprocedural effects are combined in the second step of the algorithm flow-insensitively.
3. **Optimizations to the CLA algorithm.** The second step of the algorithm applies an extended CLA algorithm on the summary graphs representing the methods to compute the interprocedural points-to information. The algorithm starts by analyzing the `main` method and analyzes only the methods that are reachable from `main`. It iterates over the summaries and the points-to relations until a fixpoint is reached. We introduce two optimizations to speed up the computation: we reuse the cache of intermediate results between iterations to minimize the storage management overhead, and we minimize the application of the expensive set union operation.

This paper presents empirical results demonstrating the efficiency of the proposed technique. Our algorithm successfully analyzes over a million lines of Java source code on the order of minutes, using only 512MB of heap space. We apply the points-to results to the problem of concrete type inferencing. The results indicate that the call graph generated is very precise: for the same call sites, it is about twice as precise as rapid type analysis. Many other applications are also made possible with a fast inclusion-based points-to analysis.

2 Sparse Intra-method Summaries

Rather than operate on the program source directly, our algorithm operates on a sparse "summary" representation generated from the source. This has a number of advantages. First, it reduces the amount of code that the interprocedural algorithm must deal with, thus improving the speed and the scalability of the analysis. Second, because each method is small, we can afford to use a flow-sensitive

technique to generate the method summary. This allows the flow-insensitive interprocedural algorithm to take advantage of some local flow sensitivity, which is essential for accurately keeping track of updates to local variables.

A method summary is a flow-insensitive summary that encodes all stores (writes) performed by the method, and all calls made by the method, along with their parameters. The summary also includes information about return values, and thrown and caught exceptions. For simplicity, we do not describe constants, exceptions, type casts, or invocations of native methods here; our implementation correctly handles these features, however.

2.1 Program Representation

Let us start with an example of a method summary. As shown in Figure 1 is a summary graph for an `add` method for a linked list. The implicit *this* parameter and the *o* parameter are represented by the nodes marked *this* and *o*, respectively. The dashed edges are access path edges, marked with field names. The solid edges are write edges, which record the possible updates that the method can make. The node marked List is a concretely-typed node that represents the object created at the `new` statement. The call to `equals` is also recorded in the summary graph, with dotted links to the nodes that represent each of the parameters.

```
class List {
  Object data; List next;
  boolean add(Object o) {
    List p = this;
    for (;;) {
      if (o.equals(p.data)) return false;
      if (p.next == null) break;
      p = p.next;
    }
    p.next = new List(); p.next.data = o;
    return true;
  }
}
```

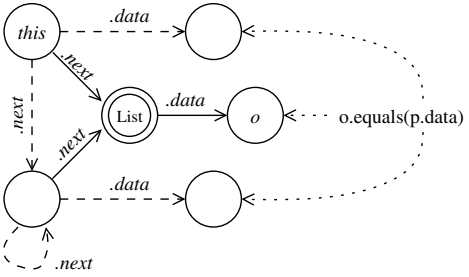


Fig. 1. Example code and summary graph.

A program is represented as follows. There is a set $\mathtt{l} \in L$ of local variables and a global variable g. The analysis models static variables as fields of the global object. Together, the local variables and the global variable form the set $v \in V = L \cup \{g\}$ of variables.

There is a set of class types $cl \in CL$, which refer to the classes in the program. There is also a set $f \in F$ of object fields, which are accessed using syntax of the form $v.f$. Static variables are accessed using syntax of the form $g.f$. There is also a set $op \in OP$ of methods. Each method has a formal parameter list p_0,...,p_k. Parameter p_0 refers to the receiver object.

Each method is represented by a control flow graph. The nodes are statements $st \in ST$. All statements are in one of the following forms:

- A copy statement $\mathtt{l} = v$
- An object creation statement $\mathtt{l} = \mathit{new}\ cl$
- A load statement $\mathtt{l}_1 = \mathtt{l}_2.f$
- A store statement $\mathtt{l}_1.f = \mathtt{l}_2$
- A method invocation statement of the form $\mathtt{l} = \mathtt{l}_0.op(\mathtt{l}_1,...,\mathtt{l}_k)$
- A return statement `return l`
- An enter statement $\mathtt{enter}_{op}$
- An exit statement $\mathtt{exit}_{op}$

The control flow graph for each method *op* starts with an enter statement $\mathtt{enter}_{op}$ before any other statements, and ends with an exit statement $\mathtt{exit}_{op}$ after any other statements.

2.2 Object Representation

The analysis represents the objects that the program manipulates using a set $n \in N$ of nodes. Nodes fall into six types:

- Set of concretely-typed nodes N_{C}. A concretely-typed node represents an object that is known to have a concrete, well-known type. There is one concretely-typed node n_{C} for each object creation statement in the program. Each concretely-typed node has an associated type $cl \in CL$, denoted by $\mathrm{cl}(n_{\mathrm{C}})$.
- Set of parameter nodes N_{P}. There is one parameter node n_{P} for each formal parameter p_k in the program; it represents the object passed in as that parameter into the method currently being analyzed. $\mathrm{k}(n_{\mathrm{P}})$ refers to the index of the parameter referred to by the parameter node n_{P}.
- Set of return value nodes N_{RV}. There is one return value node n_{RV} for each method invocation statement *st* in the program; it signifies the objects that can be returned by the invoked method.
- Set of local return value nodes N_{LRV}. There is one local return value node n_{LRV} for each method *op* in the program; it signifies the values that can be returned by that method.
- Global node, G, which holds references to static variables.
- Set of load nodes N_{L}, which represent entries in access paths off of other nodes. Each load node n_{L} has an associated field $\mathrm{f}(n_{\mathrm{L}}) \in F$, a set of base nodes $\mathrm{base}(n_{\mathrm{L}}) \subseteq N$, and an associated load statement $\mathrm{st}(n_{\mathrm{L}}) \in ST$.

The analysis treats an array as a node with a special field *elements*. The *elements* field represents all of the elements of the array. Note that a single node can correspond to multiple objects. Likewise, due to aliasing, multiple nodes may correspond to a single object at run time. Because we only perform weak updates to pointer variables and because dereferences are not resolved until the interprocedural phase, the effect is a safe, conservative approximation of the effect of the method in all possible contexts.

2.3 Summary Graphs

A summary graph consists of a set of access path (read) edges $E_{\mathrm{R}} \subseteq N \times F \times N_{\mathrm{L}}$, a set of write edges $E_{\mathrm{W}} \subseteq (V \cup N) \times F \times N$, and an outgoing parameter map $P \subseteq N \times \mathcal{N} \times ST$, where $\mathcal{N}$ represents the set of natural numbers.

Read edges are created by load statements and they refer to references that may have been created outside of the current method. Write edges are created by store statements and refer to references created by this method. Following a sequence of edges from a parameter or global node to a load node gives us the access path to that load node. The outgoing parameter map is a mapping between nodes and method call parameters. It records which nodes are passed as which parameters to the called methods.

We denote the edges in set S from a node n as $\mathrm{edgesFrom}(S, n)$. The source of an edge e is $\mathrm{from}(e)$, and the target of an edge e is $\mathrm{to}(e)$. We denote the targets of read edges from node n on field f as $E_{\mathrm{R}}(n, f)$. Likewise, the targets of write edges from node n on field f are denoted as $E_{\mathrm{W}}(n, f)$. The union of E_{R} and E_{W} is E. We denote an edge from node n_1 to node n_2 on field f as $n_1 \xrightarrow{f} n_2$; accesses of scalar variables are unlabeled. For a method call statement st and parameter number j, we denote the set of nodes that can be passed as that parameter to that method call statement as $P(st, j)$. We denote an edge from the node n, representing the jth actual parameter of the call statement st as $n \xrightarrow{j} st$.

2.4 Data-Flow Analysis

The algorithm uses a data-flow analysis to generate a summary graph at every point in the method. Initially, there are no read edges or outgoing parameters, and, there are implicit write edges between the global variable g and the global node G, and between the first n local variables and the parameters, where n is the number of parameters. That is, for a method with formal parameters $p_0, ..., p_n$, the initial summary graph for the method has read edges $E_{\mathrm{R}} = \emptyset$, write edges $E_{\mathrm{W}} = \{g \rightarrow G\} \cup \{\langle \mathtt{l}_k \rightarrow n_{p_k} \rangle \mid n_{p_k} \in N_{\mathrm{P}}\}$, and outgoing parameter map $P = \emptyset$.

Once the algorithm has generated the initial summary graph, it propagates summary graphs through the statements of the method's control flow graph, applying transfer functions and computing joins, until the result stabilizes. We use a standard work-list solver to solve the data flow. The final result is the computed summary graph after the exit statement, $\mathtt{exit}_{op}$.

A transfer function $\langle E'_{\mathrm{R}}, E'_{\mathrm{W}}, P' \rangle = [st](\langle E_{\mathrm{R}}, E_{\mathrm{W}}, P \rangle)$ models the effect of a statement st on the summary graph. The following gives the transfer function for each statement; unless specified otherwise, we assume:

$$E'_{\mathrm{R}} = E_{\mathrm{R}} \quad E'_{\mathrm{W}} = E_{\mathrm{W}} \quad P' = P$$

Copy statements. A copy statement of the form $\mathtt{l} = v$ makes $\mathtt{l}$ point to the object that v points to. The transfer function updates E_{W} to reflect this change by killing the current set of edges from $\mathtt{l}$, then generating write edges from $\mathtt{l}$ to all of the nodes that v points to.

$$\begin{aligned}\text{Kill}_{E_\text{W}} &= \text{edgesFrom}(E_\text{W}, \texttt{l})\\ \text{Gen}_{E_\text{W}} &= \{\texttt{l} \rightarrow n \mid n \in E_\text{W}(v)\}\\ E'_\text{W} &= (E_\text{W} - \text{Kill}_{E_\text{W}}) \cup \text{Gen}_{E_\text{W}}\end{aligned}$$

Object creation statements. An object creation statement $\texttt{l} = \textit{new cl}$ allocates a new object of type *cl* and makes $\texttt{l}$ point to the object. All objects that are allocated at a specific creation statement are represented by a single, concretely-typed node n_C. The transfer function kills all edges from $\texttt{l}$, then generates a write edge from $\texttt{l}$ to n_C.

$$\begin{aligned}\text{Kill}_{E_\text{W}} &= \text{edgesFrom}(E_\text{W}, \texttt{l})\\ \text{Gen}_{E_\text{W}} &= \{\texttt{l} \rightarrow n_\text{C}\}\\ E'_\text{W} &= (E_\text{W} - \text{Kill}_{E_\text{W}}) \cup \text{Gen}_{E_\text{W}}\end{aligned}$$

Load statements. A load statement *st* of the form $\texttt{l}_1 = \texttt{l}_2.f$ makes $\texttt{l}_1$ point to the object that $\texttt{l}_2.f$ points to. A load node represents the abstract location accessed by $\texttt{l}_2.f$. For each node n_2 pointed to by $\texttt{l}_2$, we create a load node n_L and add an access path (read) edge $n_2 \xrightarrow{f} n_\text{L}$.

We must be careful when creating load nodes on cyclic access paths to avoid infinite expansion of cyclic access paths. For each node n_2, we compute the transitive closure of predecessor base nodes, $\text{bases}(n_2)$. If a node in $\text{bases}(n_2)$ comes from the same load statement, we use that node rather than creating a new node.

$$\begin{aligned}n_\text{L}(n_2) &= n' \text{ where } n' \in \text{bases}(n_2), \text{st}(n') = \textit{st}, \text{ if } n' \text{ exists}\\ &\quad\ n'' \text{ where } n'' \in N_\text{L} \text{ is a fresh node, otherwise.}\\ \text{Gen}_{E_\text{R}} &= \{n_2 \xrightarrow{f} n_\text{L}(n_2) \mid n_2 \in E_\text{W}(\texttt{l}_2)\}\\ \text{Kill}_{E_\text{W}} &= \text{edgesFrom}(E_\text{R}, \texttt{l}_1)\\ \text{Gen}_{E_\text{W}} &= \{\texttt{l}_1 \rightarrow n_\text{L}(n_2) \mid n_2 \in E_\text{W}(\texttt{l}_2)\}\\ E'_\text{W} &= (E_\text{W} - \text{Kill}_{E_\text{W}}) \cup \text{Gen}_{E_\text{W}}\\ E'_\text{R} &= E_\text{R} \cup \text{Gen}_{E_\text{R}}\end{aligned}$$

Store statements. A store statement of the form $\texttt{l}_1.f = \texttt{l}_2$ makes the f field of the object referred to by $\texttt{l}_1$ refer to the same object referred to by $\texttt{l}_2$. The analysis models this by finding the set of nodes that $\texttt{l}_1$ points to, then generating write edges from all of these nodes to the nodes that $\texttt{l}_2$ points to.

$$\begin{aligned}\text{Gen}_{E_\text{W}} &= \{n_1 \xrightarrow{f} n_2 \mid n_1 \in E_\text{W}(\texttt{l}_1), n_2 \in E_\text{W}(\texttt{l}_2)\}\\ E'_\text{W} &= E_\text{W} \cup \text{Gen}_{E_\text{W}}\end{aligned}$$

Method invocation statements. Method invocation statements invoke a method with the given parameters and return a value, which is stored into the local variable $\texttt{l}$. To handle return values, every method invocation statement has a return value node n_RV, which signifies the possible return values of the method invocation. The transfer function kills all edges from $\texttt{l}$, then generates a write edge from $\texttt{l}$ to n_RV.

Our intra-method analysis also records the sets of nodes that are passed in as each parameter. This information is later used in the inter-method analysis phase. For a method invocation statement *st* of the form $\texttt{l} = \texttt{l}_0.op(\texttt{l}_1,\ldots,\texttt{l}_k)$, we record the set of nodes passed into the jth parameter, $0 \leq j \leq k$.

$$\begin{aligned}
\mathrm{Gen}_P &= \{n \xrightarrow{j} st \mid n \in E_{\mathrm{W}}(\mathtt{l}_j)\} \\
P' &= P \cup \mathrm{Gen}_P \\
\mathrm{Kill}_{E_{\mathrm{W}}} &= \mathrm{edgesFrom}(E_{\mathrm{W}}, \mathtt{l}) \\
\mathrm{Gen}_{E_{\mathrm{W}}} &= \{\mathtt{l} \rightarrow n_{\mathrm{RV}}\} \\
E'_{\mathrm{W}} &= (E_{\mathrm{W}} - \mathrm{Kill}_{E_{\mathrm{W}}}) \cup \mathrm{Gen}_{E_{\mathrm{W}}}
\end{aligned}$$

Return statements. A return statement `return l` returns the object pointed to by `l` to the caller method. We model this in the transfer function by generating write edges from the local return value node n_{LRV} to the nodes pointed to by `l`.

$$\begin{aligned}
\mathrm{Gen}_{E_{\mathrm{W}}} &= \{n_{\mathrm{LRV}} \rightarrow n \mid n \in E_{\mathrm{W}}(\mathtt{l})\} \\
E'_{\mathrm{W}} &= E_{\mathrm{W}} \cup \mathrm{Gen}_{E_{\mathrm{W}}}
\end{aligned}$$

Exit statements. The exit statement, $\mathtt{exit}_{op}$, removes all write edges from local variables $\mathtt{l} \in L$ and the global variable g, as they are no longer relevant outside of the context.

$$\begin{aligned}
\mathrm{Kill}_{E_{\mathrm{W}}} &= \textstyle\bigcup_{v \in V} \mathrm{edgesFrom}(E_{\mathrm{W}}, v) \\
E'_{\mathrm{W}} &= E_{\mathrm{W}} - \mathrm{Kill}_{E_{\mathrm{W}}}
\end{aligned}$$

Control-flow join points. The confluence operator $\underline{\sqcup}$ for points-to graphs at control-flow joins is:

$$\underline{\bigsqcup}(\langle E_{\mathrm{R}_1}, E_{\mathrm{W}_1}, P_1\rangle, \langle E_{\mathrm{R}_2}, E_{\mathrm{W}_2}, P_2\rangle) = \langle E_{\mathrm{R}_1} \cup E_{\mathrm{R}_2}, E_{\mathrm{W}_1} \cup E_{\mathrm{W}_2}, P_1 \cup P_2\rangle$$

3 Inclusion-Based Points-To Analysis

Our inclusion-based points-to analysis algorithm is essentially the algorithm for C programs due to Heintze and Tardieu, which is based on Andersen's points-to analysis. We have extended their algorithm to handle modern language features such as object fields, strict typing, virtual methods, type casts, and exceptions. We have also optimized the algorithm to use type information, reuse caches between iterations, and to reduce the number of expensive set union operations.

3.1 Review: Andersen's Points-To Analysis

An inclusion-based analysis, such as Andersen's algorithm, models points-to information as inclusion relations. That is, an assignment such as $x = y$ implies the constraint that x points to all the objects pointed to by y, written $x \supseteq y$. The assignment is uni-directional: information only flows from y to x, and not vice-versa. Heintze and Tardieu showed that Andersen's $\mathrm{O}(n^3)$ algorithm could be implemented efficiently for C programs using a graph-based algorithm with caching and online cycle detection. Our analysis and formulation are based on the presentation by Heintze and Tardieu, which was originally based on a presentation style by McAllester.

Consider a simple language L where all expressions take the form:

$$e \ ::= \ x \ \mid \ x.f \ \mid \ new_x$$

$$\frac{x \Longrightarrow new_y}{new_y.f \Longrightarrow e} \quad (\text{if } x.f = e \text{ in } Q) \quad [\textsc{Store}]$$

$$\frac{x \Longrightarrow new_y}{e \Longrightarrow new_y.f} \quad (\text{if } e = x.f \text{ in } Q) \quad [\textsc{Load}]$$

$$\frac{}{e_1 \Longrightarrow e_2} \quad (\text{if } e_1 = e_2 \text{ in } Q) \quad [\textsc{Copy}]$$

$$\frac{e_1 \Longrightarrow e_2, e_2 \Longrightarrow e_3}{e_1 \Longrightarrow e_3} \quad [\textsc{Trans}]$$

Fig. 2. Deduction rules for Andersen's analysis on program Q

Programs are sequences of assignments of the form $e_1 = e_2$, where e_1 cannot be of the form new_x. new_x corresponds to the object returned at object creation site x. Given some program Q, we use the deduction rules in Figure 2 as the constraints of the system. Conceptually, an inclusion edge $\Longrightarrow$ refers to an *inclusion relationship* between lvalues; that is, $x \Longrightarrow y$ represents the relation $x \supseteq y$.

The Store rule specifies that if a store instruction of the form $x.f = e$ exists in the program and there is an inclusion edge from x to new_y, then add an inclusion edge from $new_y.f$ to e. The Load rule specifies that if a load instruction of the form $e = x.f$ exists in the program and there is an inclusion edge from x to new_y, then add an inclusion edge from e to $new_y.f$. The Copy rule adds an inclusion edge from e_1 to e_2 for all assignments $e_1 = e_2$ in the program. The Trans rule is transitive closure.

3.2 Graph-Based Andersen's Analysis

As a flow-insensitive and context-insensitive analysis, Andersen's algorithm is imprecise and the resulting points-to sets of each location can potentially be very large. Another problem is that computing the full transitive closure of the graph is very expensive. Most of the work on speeding up Andersen's analysis goes into avoiding explicitly representing all points-to sets in the program, and avoiding computing the full transitive closure of the graph. Our algorithm is based on the pre-transitive graph algorithm by Heintze and Tardieu, which gathers its savings from intelligent caching and online cycle detection and collapsing.

Our algorithm operates on the sparse method summaries described in Section 2. It builds up a set of inclusion edges between nodes in that representation. Like in Heintze and Tardieu's algorithm, we keep the inclusion edge graph in pre-transitive form. The read and write edges in the sparse method summaries correspond to the Load and Store statements in language L, respectively. To capture the interprocedural effects, as the targets of the method invocations become resolved, our algorithm creates the equivalent of Copy statements that copy the actual parameters to the corresponding formal parameters, and from the returned values to the variables receiving the result.

The source and destination nodes in the summaries can represent *abstract locations*, for example, parameter nodes, load nodes, etc. We need to find the

```
main(){
    add summaries of root methods to methodSet
    iterationCount = 1
    do {
        noChange = true // set to false if any set changes
        foreach method summary ms in methodSet do
            foreach (write) edge e in E_W with field f in ms do
                foreach n in getConcreteNodes(from(e)) do
                    if n contains field f
                        add write edge n.f → to(e) to E_W
            foreach (read) edge e in E_R with field f in ms do
                foreach n in getConcreteNodes(from(e)) do
                    add inclusion edge to(e) ⟹ E_W(n, f)
            foreach method call st in ms
                S = { }
                foreach n in getConcreteNodes(P(st, 0)) do
                    S = S ∪ targets(st, cl(n))
                foreach method op in S do
                    add method summary of op, ms', to methodSet
                    addCallMappings(ms, st, ms')
        iterationCount++
    } until noChange
}
addCallMappings(caller, st, callee) {
    if already mapped (st, callee) return
    foreach parameter node n_P in callee do
        foreach node n in P(st, k(n_P)) in caller do
            add inclusion edge n_P ⟹ n
    foreach node n in E_W(n_LRV) in callee do
        add inclusion edge n_RV(st) ⟹ n
}
```

Fig. 3. Main algorithm

concrete nodes that these abstract nodes refer to in order to find what locations the write edges (store statements) could be writing to, and likewise what locations that read edges (load statements) could be reading from. Similarly, method calls may have abstract nodes as their receiver objects: we need to find the corresponding concrete nodes to find the possible class types and thereby the possible target methods of the method call.

We use the inclusion edge graph to find these concrete nodes. However, because we keep the graph in pre-transitive form, we need to perform a graph reachability computation whenever we want to know the reachable concrete nodes. The *getConcreteNodes* method, described in Section 3.3, implements this graph reachability computation.

Figure 3 presents the main algorithm. We control iteration through a global *noChange* flag. This flag is set to false if any edges are added in each iteration of

the algorithm. (Note that edges are only added and therefore sets can only grow. The number of nodes is finite and so the algorithm is guaranteed to converge.) We have a set of method summaries that the algorithm iterates over; this set is initialized to the root set of the program and contains the summaries of the methods in the reachable call graph at any given point in the algorithm. We also keep track of the iteration number; this is used in versioning of the caches, as described in Section 3.3.

For each method summary, we visit every write edge, every read edge, and every method invocation. For each write edge, we find the concrete nodes that correspond to the source node of the edge. Then, we add write edges on the same field from each of the concrete nodes to the target of the edge. However, we do not add write edges from concrete nodes that do not have the named field. This is one example where having a strictly-typed language allows us to refine the points-to information.

Java contains many virtual method invocations that depend on the type of the receiver object. We use the points-to information and the strict type information to find the possible targets of virtual method invocations and bind their parameters and return values. For each method call in the summary, we build up a set of the class types of the concrete nodes of the receiver object. We use that set to find the possible target methods of that method invocation statement and add the target methods to the method set. The function *targets*(*st*, *cl*) returns the set of possible methods invoked by statement *st* in the case the receiver node is of class *cl*. The *addCallMappings* function is used to insert inclusion edges due to method invocations. *addCallMappings* skips call site and target pairs that have already been mapped. It adds inclusion edges from each formal parameter node in the callee to the actual parameters from the parameter map in the caller. It also adds inclusion edges from the return value node of that method invocation in the caller to the returned nodes of the callee.

3.3 Reachability Computation

We adopted the online cycle detection mechanism of Heintze and Tardieu's algorithm in our reachability computation on the inclusion edge graph, as shown in Figure 4. *getConcreteNodes* is a wrapper function for *getConcreteNodes**, which takes two arguments (a node to explore from and a list of nodes that defines the path being explored) and returns two results (the set of concrete nodes that are reachable from the first argument and a boolean value stating whether or not the set of nodes has changed since the last iteration). *getConcreteNodes* passes **null** as the path to the recursive *getConcreteNodes** function and returns the set of reachable concrete nodes to its caller.

To detect cycles in the path, each node has a boolean flag called *onPath* to mark if the node is already on the path being explored. When a cycle is detected, *getConcreteNodes** unifies the nodes in the cycle. The function *unify* unifies two nodes. Node unification is implemented by an optional *skip* field for each node. Two nodes n_1 and n_2 are unified by setting $n_1.skip$ to n_2, and adding all outgoing inclusion edges from n_1 to n_2. Subsequently, whenever node n_1 is

```
getConcreteNodes(n) {
    (r, changed) = getConcreteNodes*(n, null)
    return r
}
getConcreteNodes*(n, path) {
    if (n ∈ N_C) return ({n}, false)
    if (n.onPath) { // cycle detected
        while (car(path) ≠ n)
            unify(car(path), n); path = cdr(path)
        remove cyclic edges from n
        return ({ }, false)
    } else {
        (r, changed, ncount) = cache.get(n)
        if (r == null) // does not exist in cache
            r = { }
        else if (ncount == iterationCount)
            return (result, changed)
        path = cons(n, path)
        n.onPath = true; changed = false
        foreach inclusion edge n → n' do
            (r2, changed2) = getConcreteNodes*(n', path)
            if (changed2 || n → n' never visited)
                r = r ∪ r2; if r grows, changed = true
        n.onPath = false
        cache.put(n, r, changed, iterationCount)
        return(r, changed)
    }
}
```

Fig. 4. *getConcreteNodes*: graph reachability

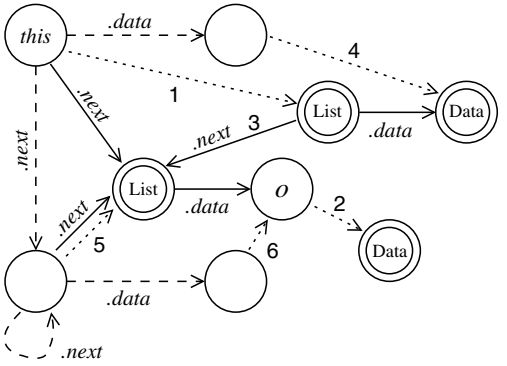
accessed, we follow its *skip* pointer. After unifying a set of nodes, we eliminate transitive self-edges. This method of cycle elimination is very low cost, yet it finds and collapses all cycles in the parts of the graph that we traverse.

The results of the calls to *getConcreteNodes** are cached for efficiency. The reachability of a node is guaranteed to be computed accurately only if it is the first such request in an iteration of the main algorithm. The same iteration is likely to need the reachability information of the same node over and over again, and the true answer may change during the iteration as more edges are added. For efficiency, *getConcreteNodes** returns the cached result, if one has been computed for the current iteration, and relies on future iterations to propagate the correct answer.

Heintze and Tardieu's algorithm also caches the results, but deletes them after every iteration. We noticed from our experiments that many of the cached results did not change between iterations and that a significant amount of time was spent reallocating and rebuilding these caches on every iteration. Therefore,

we keep the cache entries across iterations, but store along with each entry the iteration number in which it was last computed and whether the last recomputation changed its value. If the iteration number on the entry is not current, its value is recomputed. The function *cache.get*(n) returns the triple (concrete node set, changed flag, iteration number). Likewise, *cache.put* updates the cache. We use soft references for the cache entries so that old entries can be flushed when memory is low.

Set union operations are expensive. We minimize the number of set union operations performed by skipping the $r = r \cup r_2$ step whenever we can determine that r_2 has not changed since the last computation of r. More specifically, we check if r_2's value in the cache is current, i.e. it has not changed since the last iteration, and that we have visited that edge before. We cache the results, along with the updated iteration number and whether the set has changed.



Here is the result after one iteration of our algorithm operating on the summary graph from Figure 1, when the `add` method is called on a single-element list. The three rightmost concrete nodes correspond to the single-element list and the element to add. Inclusion edges are marked as dotted edges. Edges are added by the algorithm in numerical order. Edges 1 and 2 are from parameter mapping, edge 3 is from the callee write edge, and edges 4-6 are from callee read edges.

Fig. 5. Example of interprocedural algorithm on the graph from Figure 1.

4 Experimental Results

The inclusion-based points-to analysis described above was implemented in the joeq system[12] in about 800 lines of source; the implementation is available as part of the open-source joeq project. The experiments were performed on a PC with a 2GHz Pentium 4 and 2 GB of memory, running Redhat Linux 7.2 and Java HotSpot VM build 1.3.1_01. We ran all tests with a heap size of 512MB, even though only some of the benchmarks ended up needing that much memory.

Figure 6 shows the results of our analysis on a variety of benchmarks. The first nine benchmarks are from the standard SpecJVM v1.03 benchmark suite. The next six benchmarks are components of the reference implementation of Java 2 Enterprise Edition version 1.3 (J2EE). With over 900,000 lines of code,

Benchmark	Classes	Methods	Byte-codes	Calls	Ave. Call Targets RTA	Ave. Call Targets Points-to	Time (s) Opt	Time (s) No opt	Space (MB)	Itera-tions
check	202	1342	112K	9428	1.79	1.02	14	47	18	22
compress	92	304	27K	1408	2.32	1.01	5	49	3	16
db	95	360	30K	1730	2.47	1.01	5	52	4	13
jack	139	573	57K	3468	2.05	1.01	3	59	1	15
javac	262	1015	89K	5955	2.58	1.17	13	115	14	18
jess	306	1654	129K	10723	1.85	1.02	16	69	27	22
mpegaudio	243	1529	158K	9852	1.79	1.02	16	61	25	22
mtrt	84	254	22K	1141	3.00	1.01	4	45	2	15
raytrace	84	254	22K	1140	3.00	1.01	4	45	2	15
admintool	283	1563	108K	6628	2.73	1.22	19	155	28	18
appclient	847	5037	342K	26888	2.29	1.18	318	960	285	29
deploytool	1231	8194	532K	40156	2.38	1.43	744	1788	507	33
j2eeserver	1021	6039	413K	32564	2.43	1.39	516	2479	494	34
packager	355	1869	143K	10748	2.04	1.05	28	481	26	26
verifier	753	4750	336K	25810	1.99	1.07	74	1101	76	24
joeq	1229	9890	450K	40151	2.63	1.34	136	372	193	12
cloudscape	859	8166	483K	35493	1.95	1.03	211	612	70	22
jedit	1371	10365	691K	51656	2.36	1.30	614	1953	472	24

Fig. 6. Analysis results on the benchmarks

J2EE is a sizable framework for building Java Enterprise applications. joeq is the joeq virtual machine, upon which the analysis was implemented. It has about 75,000 lines of code. Cloudscape is a database server that is shipped with J2EE. The source code is not available. jEdit, containing over 100,000 lines of code, is a full-featured editor written in Java.

The first four columns of numbers in the table specify the number of classes, methods, bytecodes, and call sites that were analyzed. Our algorithm analyzes only those methods reached through the root set, including code in the class libraries; the numbers reflect this reachable code. We found that the algorithm was successful in greatly reducing the amount of code that need to be dealt with.

The next two columns give the average number of targets per call site when using a basic RTA analysis [2] and when using our analysis. Our analysis is effective in reducing the number of targets to almost 1, in many cases. The next two columns give the run time of our algorithm, with and without our optimizations described in Section 3.3. By reducing the number of potentially $O(n^2)$ set union operations for large points-to sets, the analysis speeds up significantly, by a factor between 2.4 and 18.3. The last two columns give the amount of heap space used for the analysis and the number of iterations taken, respectively. The algorithm uses a reasonable amount of memory and converges fairly quickly. Our algorithm analyzed over 1200 classes, 8000 methods, and 500KB of byte codes in 12 minutes, in Java nonetheless.

We also ran our analysis in a field-independent manner (i.e. by ignoring all field names.) When in field-independent mode, the run times skyrocketed. Compress, one of the smallest benchmarks in the suite, took more than two hours to analyze. We did not attempt any of the larger benchmarks. This shows that field sensitivity is essential when running the algorithm on Java code.

5 Related Work

We have adapted and extended Heintze and Tardieu's Andersen-style points-to analysis designed originally for C programs to work on Java programs. The most significant difference is that our algorithm is field-sensitive. By tracking the different fields in an object separately, our algorithm is both faster and more precise than a field-independent technique. In addition, our algorithm is locally flow-sensitive, allowing local pointers to be destructively updated. Our algorithm handles all of the Java constructs which include virtual method invocations, type casts, exceptions, unknown and escaping pointers. Finally, we have made a couple of practical optimizations to improve the algorithm's efficiency.

Among previous work that extends Andersen's algorithm to Java[6,7,9], the algorithm by Rountev et al. appears to be the fastest and the most precise[7]. Like our algorithm, their technique is field-sensitive. They adapted and extended a constraints-based algorithm[4,10] also originally designed for C to Java. Of all the experimental results reported, `javac` took the longest to run, requiring 6 minutes on a 360-Mhz Sun Ultra60 machine with 512 MB of memory. Our algorithm is more precise because it is locally flow-sensitive. By using a CLA-based technique, our algorithm is also faster, taking 12 seconds on `javac` on a 2GHz PC. We have also reported results on much larger programs. Liang et al. experimented with a number of different context-insensitive and flow-insensitive techniques based on Steensgaard's and Andersen's algorithms[6]. Like Heintze and Tardieu, they experimented with a field-independent and a field-based analysis. They observed context insensitivity does not provide very accurate results when applied to collections and maps. They suggested representing these objects with special models that allow some degree of context sensitivity. They recommended the use of a field-based Andersen algorithm as the best tradeoff between time and precision. In view of our results, there is no reason not to use a more precise field-sensitive technique.

6 Conclusion

We have presented an efficient inclusion-based points-to analysis for Java, based on the Heintze and Tardieu's CLA algorithm originally designed for C. Our algorithm handles all the Java language features including inheritance, object fields, exceptional control flow, type casting, dynamic dispatch, and reflection.

We have extended the CLA algorithm in two ways to improve its precision. First, our analysis is field-sensitive. Fields in an object are treated distinctly

from each other. Second, our analysis handles local variables in a method flow-sensitively; this is important because locations on the stack are often reused in Java bytecodes to hold different objects. Our algorithm uses a data-flow analysis to compute a summary graph for each program point in the program. Only weak updates are allowed on non-local variables; the summary graph at the exit of a method is a flow-insensitive representation of the local effect of a method. By improving the algorithm's precision, these two techniques also improve its efficiency due to the smaller call graphs produced.

We have also proposed two techniques to improve the efficiency of the CLA algorithm. We reuse the data in the cache to minimize the memory management overhead, and we avoid expensive set union operations if the operands have not changed. This paper presents empirical results demonstrating that our algorithm is scalable to extremely large code bases without sacrificing accuracy.

References

1. L. Andersen. *A Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, 1994.
2. D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 324–341, 1996.
3. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP)*, 1995.
4. M. Fahndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 85–96, 1998.
5. N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 146–161, 2001.
6. D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 73–79, 2001.
7. A. Rountev, A. Milanova, and B. Ryder. Points-to analysis for Java based on annotated constraints. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 43–55, 2001.
8. B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 32–41, 1996.
9. M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an empirical comparison. Technical report, University of Passau, Sept. 2000.
10. Z. Su, M. Fahndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 81–95, 2000.
11. V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 264–280, 2000.
12. J. Whaley. joeq virtual machine. http://joeq.sourceforge.net, 2001.

Compactly Representing First-Order Structures for Static Analysis

R. Manevich[1,⋆], G. Ramalingam[2], J. Field[2], D. Goyal[2], and M. Sagiv[1]

[1] Tel Aviv University
{rumster,msagiv}@post.tau.ac.il
[2] IBM T.J. Watson Research Center
{rama,jfield,dgoyal}@watson.ibm.com

Abstract. A fundamental bottleneck in applying sophisticated static analyses to large programs is the space consumed by abstract program states. This is particularly true when analyzing programs that make extensive use of heap-allocated data. The TVLA (**T**hree-**V**alued **L**ogic **A**nalysis) program analysis and verification system models dynamic allocation precisely by representing program states as *first-order structures*. In such a representation, a finite collection of *predicates* is used to define states; the predicates range over a universe of individuals that may evolve—expand and contract—during analysis. Evolving first-order structures can be used to encode a wide variety of analyses, including most analyses whose abstract states are represented by directed graphs or maps. This paper addresses the problem of space consumption in such analyses by describing and evaluating two novel structure representation techniques. One technique uses *ordered binary decision diagrams* (OBDDs); the other uses a variant of a *functional map* data structure. Using a suite of benchmark analysis problems, we systematically compare the new representations with TVLA's existing state representation. The results show that both the OBDD and functional implementations reduce space consumption in TVLA by a factor of 4 to 10 relative to the current TVLA state representation, without compromising analysis time. In addition to TVLA, we believe that our results are applicable to many program analysis systems that represent states as graphs, maps, or other structures of similar complexity.

1 Overview and Main Results

A fundamental bottleneck in applying sophisticated static analyses to large programs is the space consumed during analysis by abstract program states. This is particularly true for modern programs where much of the data manipulated by the application lies in the heap (i.e., in dynamically allocated data structures), instead of being stored in a fixed set of variables. An abstraction of the potentially unbounded heap that is precise enough for a particular analysis may often

⋆ Partially supported by the Israeli Academy of Science. Part of this work was done while visiting IBM's T.J. Watson Research Center.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 196–212, 2002.
© Springer-Verlag Berlin Heidelberg 2002

have prohibitive space requirements. The TVLA (**T**hree-**V**alued **L**ogic **A**nalysis) program analysis and verification system [6] is designed to model dynamic allocation precisely by representing program states as *first-order structures*. In such a representation, a finite collection of *predicates* is used to define states; the predicates range over a universe of *individuals* that may evolve—expand and contract—during analysis. The use of first-order structures permits, e.g., dynamic memory allocation or dynamic thread creation to be modelled in a natural way [13,17]. In addition, first-order state representations can be much more compact than representations based on finite-state models. For example, when representing programs with pointers, a natural finite state representation of pointer aliases usually consumes quadratic space [14], whereas a graph based (first-order) representation such the one used by TVLA can be linear in many cases.

TVLA has been successfully applied to a wide variety of deep program analysis and verification problems, including checking C programs for the presence of memory leaks and dangling or uninitialized pointers [4], verifying the correctness of a simple garbage collector [12], determining whether clients of a Java library satisfy the library's *conformance constraints* for correct usage [11], and verifying certain safety properties of a packet router [10]. In carrying out these tasks, TVLA utilizes an abstract interpretation that maintains detailed information on the state of dynamically allocated memory. While this degree of precision is crucial to avoid reporting an excessive number of "false alarms", it comes at a price: the space required to analyze programs of more than a few thousand lines of code is often prohibitive.

This paper addresses the problem of space consumption in first-order state representations by describing and evaluating two new structure representation techniques. The first representation uses an existing *ordered binary decision diagram* [2,18] (OBDD) implementation [15] to encode first-order structures. The second state representation combines ideas from efficient implementations of *functional maps* (where a map derived by update to another map shares substructures of the initial map) with *normalization* via *hash-consing*. Both of these core data structures are well-known, and OBDDs have been used in several program analysis systems to represent automata or finite sets. However, it is not obvious that they should be adaptable to, or beneficial for, representing evolving first-order structures, especially given their poor worst-case behavior.

Happily, our evaluation of the new state representations indicates that they can reduce TVLA's space consumption by a factor of 4 to 10 without compromising time performance. In addition, as the number of structures manipulated by the analysis increases, the relative advantage of the new representations also increases. Since the notion of evolving first-order structure is powerful enough to encode a number of nontrivial state representations (e.g., those based on directed graphs or maps), we believe that our results are also likely to be applicable in a wide variety of program analysis techniques requiring such structures.

2 TVLA Primer

In this section, we give a brief overview of those aspects of the TVLA system that pertain to state manipulation. Complete details of the system may be found in [6].

2.1 Program States as Three-Valued Logical Structures

Given a collection of finite-arity predicates, a *first-order logical structure* is a collection of *individuals* (nodes), along with an *interpretation* (truth value assignment) of each predicate for every tuple of individuals in the predicate's domain. In order to bound the number of distinct first-order structures generated during analysis and thus ensure termination, TVLA must sometimes *blur* a structure by merging several individuals together. Blurring typically yields a *three-valued* logical structure, in which the third value, 1/2, denotes facts which may have been either 0 (false) or 1 (true) in the state prior to blurring. Logical formulas used to update or query the state are then interpreted appropriately over a three-valued structure. In general, each 3-valued logical structure represents a (possibly infinite) *set* of 2-valued structures. Nodes in a 3-valued structure that may represent more than one individual from a given 2-valued structure are called *summary nodes*. The set of individuals comprising a 3-valued structure S is called its universe (denoted by U^S); $p^S : (U^S)^k \rightarrow \{0, 1, 1/2\}$ denotes the interpretation of predicate p of arity k in structure S. (When depicting an interpretation function, we will usually omit the superscript denoting a structure when it is evident from context.)

Consider the program depicted in Fig. 1, which prints all the elements of a singly-linked list. As an expository example, let us say that we wish to confirm the (here obvious) assertion that the `printf` statement is never executed when the variable `x` has the value `NULL`. A very simple heap *shape analysis* [13] can be used for this purpose. The predicates used to define the shape analysis operate on individuals representing list elements. Pointer variables will be represented by unary predicates; for example, $y^S(u) = 1$ if the variable `y` points to the list element represented by u in structure S. Pointer *fields* within the list elements will be represented as binary predicates, thus $n^S(u_1,\ u_2) = 1$ if the `n`-field of u_1 points to u_2 in structure S. (Although the first-order machinery of TVLA is not really necessary for this simple example, it is crucial for more complicated examples, e.g., for proving that a dereferenced pointer *field* is not `NULL`.)

In TVLA, predicates are required to have arity ≤ 2; hence it is natural to depict structures in the form of directed graphs. Consider the 3-valued structure S_0 depicted in Fig. 3. Here, the variable `y` is represented by a unary predicate y, which has value 1 only for u_1. In general, a unary predicate p is represented graphically by a solid arrow connecting p to each individual u for which $p(u) = 1$, a dotted arrow if $p(u) = 1/2$, and no arrow otherwise. If p is 0-valued for all individuals, the predicate name p is not depicted; e.g., in S_0, the absence of predicate x indicates that variable `x` is null in S_0. A solid directed edge labeled by p from u_1 to u_2 denotes the fact that $p(u_1, u_2) = 1$; a dotted edge denotes

```
/* list.h */
typedef struct node
{
  struct node *n;
  int data;
} *L;
```

(a)

```
/* print.c */
#include "list.h"
void print_all(L y) {
  L x;
  x = y ;
  while (x != NULL) {
      /* assert x != NULL */
      printf("elem=%d ", x->data);
      x = x->n;
  }
}
```

(b)

Fig. 1. (a) Declaration of a linked-list data type in C. (b) A C function that prints all the elements of the list pointed to by parameter `y`.

a 1/2 value. TVLA uses a special unary predicate *sm* to maintain summary node information: $sm(w) = 1/2$ for a summary node w if w may represent multiple nodes in a corresponding 2-valued structure; otherwise, $sm(w) = 0$ if w represents a unique node. Given these conventions, we see that S_0 may be thought of as representing an infinite number of concrete data structures, including all lists of length 2 or more pointed to by program variable `y`, as well as lists with cyclic and shared tails, or lists with collections of unreachable ("garbage") elements.

```
initialize(S0) {
    for each S ∈ S0 {
        push(stack,[P_entry, S])
} }
```

```
explore() {
  while stack is not empty {
    [P, S] = pop(stack)
    if add(stateSpace(P), immutableCopy(S)) {
      for each outgoing CFG edge (P, P') {
        S' = copy(S)
        Apply (ActionSequence(P, P'), S')
        push(stack,[P', S'])
} } } }
```

Fig. 2. The structure of TVLA's abstract interpretation algorithm.

2.2 Abstract Interpretation in TVLA

TVLA takes as input a control flow graph (CFG) of the program to be analyzed—each edge of which is annotated with a sequence of *actions*—and an abstract representation of a set of initial states. Each action is an operation on the abstract data type (ADT) used to represent first-order structures (the full set of actions is enumerated in Section 3), and the action sequences associated with CFG edges collectively encode those aspects of the program's semantics that are relevant to the program analysis problem at hand. The TVLA user specifies action sequences using a high-level programming language whose core constructs are based on

first-order logic extended with transitive closure. During analysis, the high-level TVLA operations are interpreted, and translated into action sequences.

The most important high-level TVLA construct is the *predicate update* operation, which is used to encode a state update. For example, the statement `x = x->n` in Fig. 1 is modelled using a predicate update operation of the form $x(v) := \exists v_1 : x(v_1) \wedge n(v_1, v)$. This update operation is translated into a loop containing structure ADT actions whose effect is to evaluate the right-hand side of the update operation for each possible binding of the free variable v to an individual in the current structure, and to bind the result to v in the interpretation of x in a new copy of the structure.

TVLA carries out abstract interpretation by exploring all abstract program states derivable from the initial set of states via action sequences associated with CFG edges, using the algorithm outlined in Fig. 2. The actions which may be associated with each CFG edge are enumerated in Fig. 4, and described in greater detail in Section 3. The action sequences associated with edges are always terminated by actions that carry out a *blur* operation, and therefore only blurred structures are added to the state space. (The blur operation merges nodes that have the same *canonic* name together, where a node's canonic name is defined to be the sequence of values it has for the unary predicates. This guarantees that a blurred structure can have at most one node with a given canonic name.) Termination of the algorithm is guaranteed by the fact that there are only finitely many blurred structures.

The structures generated by abstract interpretation of the first iteration of the loop body of the `print_all` function are depicted in Fig. 3. The analysis begins with the 3-valued structure S_0. The actions modelling the statement `x=y` have the effect of setting `x` to point to u_1, resulting in the structure S_1. The most interesting case is the assignment `x = x->n`. This statement is modeled by the predicate update action for `x = x->n` described above, preceded by a *focus* action. The focus action, which we will not describe in detail here, has the effect of bifurcating the incoming structure into a *set* of structures; the basic idea is to replace a structure in which a predicate p has value 1/2 by a pair of structures, one where p has value 1, and one where p has value 0. Focus allows us to do a precise "case analysis" on incoming structures, which frequently sharpens the analysis results. Here, for example, focus allows us to distinguish the case where `x->n` is null in the input structure, producing an output structure where `x` is null (represented by structure $S_{2.0}$) from the input structures where `x->n` is non-null.

After two iterations, the abstract interpretation of the `print_all` function reaches a fixpoint, with an output state containing the same set of structures ($\{S_{2.0}, S_{2.1}, S_{2.2}\}$) generated after the first iteration.

3 Evolving First-Order Structures as an Abstract Data Type

In the remainder of the paper, we will describe and evaluate space-efficient representations for evolving first-order structures. These representations exploit various opportunities for space savings, most of which arise in conjunction with a

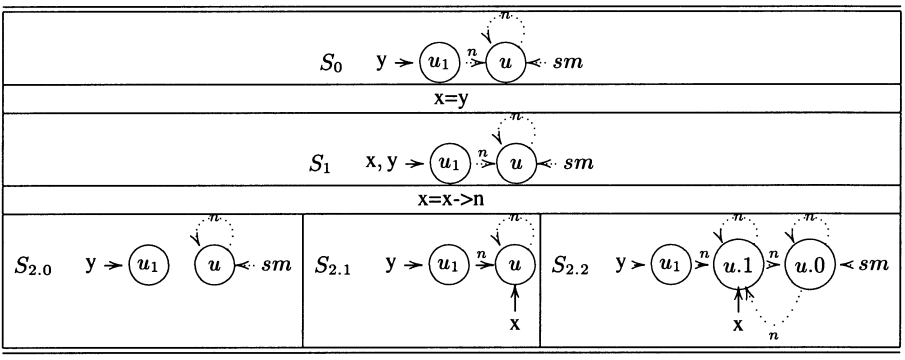


Fig. 3. Structures generated by abstract interpretation of the first iteration of the loop in the `print_all` function.

variety of abstract state representations (including, but not limited to, first-order structures). These opportunities include:

- *Sparse* data structures. For many applications, much of the state space accessible at a particular program point represents trivial values (e.g., null pointers). This fact can be exploited by avoiding explicit representation of such values.
- *Sharing* of inherited substates. The TVLA actions that model each program statement typically affect only a small portion of the program state. State representations that allow the unchanged, "inherited" portion of the state to be shared between the pre- and post-update state are therefore advantageous.
- *Normalized* (or canonical) representations for substates. Many program states contain substructures that are semantically equivalent, even though the substructures were generated by unrelated sequences of state updates. By using a canonical representation (e.g., a hash-consed tree data structure) for substates, such "serendipitous" similarity can be recognized and exploited to allow the substate to be shared. Another advantage of a canonic representation is that it enables efficient equality-checking of states, via a single pointer check, regardless of the size of the state representation. One reason why equivalent substructures often arise is that each program point typically has a set of invariants that hold true for all states associated with that point. Such invariant properties often give rise to equivalent substructures, which can in turn produce significant opportunities for space saving. This is particularly true for program points inside loops, where differing states will often contain loop-invariant substructures.
- *Phase-sensitive* state representations. Structures are initially *mutable*, and may be updated as a result of applying TVLA actions. Subsequently, they become immutable, at which point they may be compared for equality with other structures, but may not be updated. The system can exploit these phase distinctions, e.g., by using a representation that admits time-efficient

update for mutable structures, and a more space-efficient representation for immutable structures.

Before describing the different representations we study, we first present the full signature of the abstract data type (ADT) used to represent evolving first-order structures in TVLA (see Fig. 4). This signature describes the set of operations that any implementation must support.

```
type Kleene /* 0, 1, or 1/2 */
type Node /* individual in structure */
type NullaryPredicate
type UnaryPredicate
type BinaryPredicate
type TVS /* mutable three-valued structure */
type ImmutableTVS /* immutable, blurred, three-valued structure */
type TVS_SET /* set of three-valued structures */

/* evaluate predicate value for specified node(s) in TVS */
eval: TVS * NullaryPredicate -> Kleene
eval: TVS * UnaryPredicate * Node -> Kleene
eval: TVS * BinaryPredicate * Node * Node -> Kleene

/* update predicate to specified Kleene value for specified node(s) in TVS */
update: TVS * NullaryPredicate * Kleene -> void
update: TVS * UnaryPredicate * Node * Kleene -> void
update: TVS * BinaryPredicate * Node * Node * Kleene -> void

empty_TVS: TVS /* a TVS with empy universe */
empty_SET: void -> TVS_SET /* returns a new, empty, set */
copy: TVS -> TVS /* copy mutable TVS */
immutableCopy: TVS -> ImmutableTVS /* returns a blurred, immutable copy */
universe: TVS -> (Set of Node) /* enumerate nodes in TVS's universe */
new: TVS -> Node /* add a node to the TVS's universe */
remove: TVS * Node -> void /* remove node from TVS */
add: TVS_SET * ImmutableTVS -> bool /* add TVS to set */
```

Fig. 4. ADT for evolving first-order structures.

Note that the ADT contains two distinct representations of structures: a mutable structure (`TVS`) and an immutable blurred structure (`ImmutableTVS`). In addition, it defines an interface to a set of structures (`TVS_SET`), which contains only immutable blurred structures. The `add` operation for `TVS_SET` returns true if the structure being added is not an element of the set, and false otherwise; this entails checking whether the added structure is *isomorphic* to any structures already in the set. (The isomorphism check can be done in polynomial time for *blurred* structures since a blurred structure has at most one node with any given canonic name.)

The distinction between mutable and immutable structures is noteworthy. Mutable structures require a representation that allows operations such as predicate updates, and addition and removal of nodes to be done efficiently. On the other hand, the only operation performed on an immutable structure is the isomorphism test implicitly required by the `add` operation of `TVS_SET`. This distinction allows us to use a normalized, space-efficient representation for immutable

structures. This property is especially important because immutable structures last for the duration of the entire analysis, whereas the mutable structures can be discarded after they are popped from the stack and processed (See Fig. 2).

4 TVS Representations

In this section, we describe three different implementations of the TVS ADT.

4.1 Base Representation

We first describe the representation used in the public release of TVLA (TVLA version 0.91). We will refer to this as the Base representation, and use it as a baseline against which we compare the other, newer, representations.

The value of an unary (or binary) predicate p in a structure is represented by a HashMap (a hashtable based implementation of a map available in the Java Collections Framework) that contains an entry for every node i (or node pair (i, j) in the case of binary predicates) for which the value of $p(i)$ (or $p(i, j)$ respectively) is nonzero. The predicate p itself is said to have a *nonzero* value if the value of $p(i)$ (or $p(i, j)$ in the case of binary predicates) is nonzero for at least one node i (or node pair (i, j)). The value of a nullary predicate is represented by a single Kleene value.

The value of a structure itself is represented by a HashMap (which we will refer to as the first-level map) that contains an entry for every predicate p that has a nonzero value. The entry corresponding to a predicate p contains the value of p (a HashMap for non-nullary predicates and a Kleene value for nullary predicates), as well as a boolean flag, which is used to achieve some amount of sharing between the representations of different structures as explained below.

When a structure is copied, its first-level map is duplicated, but the HashMaps representing the values of (non-nullary) predicates are shared by the original and new structure. The boolean flag associated with these predicate values is set to indicate this sharing. When the value of a predicate with a shared representation needs to be updated, the underlying shared HashMap is duplicated before the update is performed. The associated boolean flag is also reset at this point to indicate the absence of sharing.

The universe of a structure is implemented using a HashSet (a hashtable based implementation of a set) plus a boolean flag to implement a similar "copy-on-write" scheme for the universe also.

Note that this representation is not a completely naive representation: it exploits sparsity and a limited form of sharing, which yield significant space savings.

4.2 OBDD Representation

We now describe a new implementation of the TVS ADT, which we will refer to as the OBDD representation. This is a phase-sensitive representation, where mutable structures are represented using the Base representation, but immutable

structures are represented using OBDDs as explained below. Due to lack of space, we will assume that readers are familiar with OBDDs, which may be used to represent boolean functions over a set of boolean variables.

Our representation uniformly models all predicates as if they were binary predicates. A unary predicate p is represented as if it were a binary predicate by translating references to $p(u)$ into a reference to $p(0, u)$. Nullary predicates are modelled using binary predicates by translating references to p into references to $p(0, 0)$. The representation uses a set P of boolean variables to identify predicates, a set N_1 of boolean variables to identify the first argument (a node) of the predicate, a set N_2 of boolean variables to identify the second argument (a node) of the predicate, and a special boolean variable $v_{1/2}$, used to extend the representation to 3-valued logic, as explained below. A 3-valued structure may then be thought of as a boolean function over these variables, one that returns the value of the predicate identified by P for the node tuple identified by N_1 and N_2. Since the value of the predicate is a Kleene ($\{0,\ 1,\ 1/2\}$) rather than a boolean, the variable $v_{1/2}$ is used to encode the third value. We represent the 3-valued structure using a corresponding OBDD with the variable ordering N ¡ P ¡ $\{v_{1/2}\}$, where N denotes the sets of variables N_1 and N_2 interleaved.

One of the goals of the representation is to ensure that (*blurred*) *isomorphic* structures end up with the *same* OBDD representation. We ensure this as follows. Recall that a node's canonic name is defined to be the sequence of values of the unary predicates for that node. When the OBDD representation of a structure needs to be created, the nodes of the structure are first sorted by canonic name and then contiguously numbered from 0 on. This guarantees that isomorphic structures will be represented by the same OBDD. Since only immutable structures are represented with OBDDs, the sets of variables N_1 and N_2 actually needed to identify nodes is determined when a structure is converted to an OBDD representation by choosing $|N_1| = |N_2| = \ulcorner \log |U^S| \urcorner$ OBDD variables.

Fig. 5 shows the OBDD representation of the structures S_0, S_1 and $S_{2.2}$ from the running example (see Fig. 3). The OBDD nodes labelled $u1$ or $\langle u, u.1\rangle$, for example, represent corresponding node tuples. Paths from the root to these OBDD nodes correspond to the node tuple boolean variables. Paths from these OBDD nodes to the terminals 0 and 1 correspond to the predicate variables. For example, the path 0101 starting from S_0 conveys the fact that in S_0 the predicate x evaluates to 0 on for the node u_1. The path 0011 starting from S_1 and stopping at the 1/2 node conveys the fact that the *sm* predicate evaluates to 1/2 on the node pair (u, u) in the structure S_1.

This figure illustrates the two kinds of sharing described earlier. The difference between the OBDD representing S_0 and the OBDD representing S_1 are the two grayed nodes, which follow the path 0101 starting from S_0. This reflects that the only difference between the structures is in the interpretation of the predicate x for the node u_1, leading to *inherited* sharing between the two OBDDs. Non-inherited sharing can be seen by observing that the node annotated by $u, \langle u, u\rangle$ is shared by both the OBDD of S_1 and $S_{2.2}$. This reflects the fact that the node u that represents the tail of the linked list in S_1 and the node $u.0$ that represents the tail of the list in $S_{2.2}$ have the same canonic name. This allows the

two structures to share the values of sm and n for this node. This sharing could easily be missed by implementations relying solely on inheritance-based sharing. The more a structure S is mutated to produce a structure S' less the inherited sharing between them, even if they contain many similar sub-structures. The OBDD representation is insensitive to the scenario by which S' evolved from S and therefore manages to exploit these similarities. Also notice that this program has a simple loop invariant that leads to sharing in the OBDD representation of the structures arising after the statement `x = x->n` ($S_{2.0}$, $S_{2.1}$ and $S_{2.2}$.) In every iteration of the loop, the node u_1, which represents the head of the list has the predicate values $x(u_1) = 0$, $y(u_1) = 1$, $sm(u_1) = 0$ and $n(u_1, u_1) = 0$, which enables all of the structures to share these values.

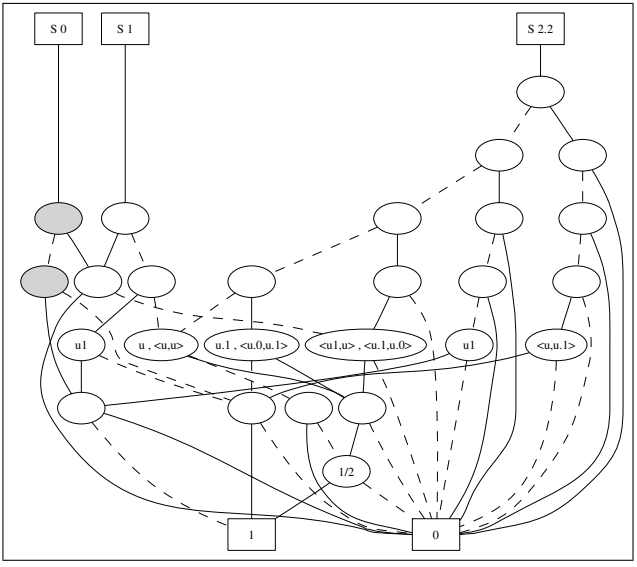


Fig. 5. The OBDDs representing the 3-valued structures in Fig. 3. Dotted edges denote 0, and solid edges denote 1. The node ordering imposed by the canonic names is $u < u_1$ for S_0 and S_1, and $u.0 < u.1 < u_1$ for $S_{2.2}$. The predicates are numbered as follows: $sm : 0, x : 1, y : 2, n : 3$.

4.3 A Functional Representation

We now describe a representation of 3-valued first order structures we refer to as a functional representation. This implementation utilizes techniques similar to those used in OBDDs, but in the context of a different data structure, namely maps. This makes it more convenient, for instance, to implement the higher level TVLA operations, without having to encode them in terms of OBDD operations.

We assume that the nullary predicates are numbered from 0 to n_0, that unary predicates are numbered from 0 to n_1, and that all binary predicates are numbered from 0 to n_2. We assume that every node in a structure is assigned a unique integer value. However, unlike in the case of predicates, the representation places no limit on the number of nodes in a structure. Further, since the set of nodes in a structure can change dynamically, the nodes in a given structure are not required to be assigned contiguous numbers. The set of nodes in a structure is implemented as a linked list, which is manipulated in a functional style to allow sharing between the representations of different structures.

The values of nullary predicates in a structure are represented by a map from integers in the range $[0:n_0]$ to Kleene. We use a tree-based functional data structure[1], which we will refer to as a *flik*, to implement such a map. A flik is either a *leaf*, capable of storing upto some fixed number l of Kleene values, or a *branch*, consisting of a fixed number a of children (each of which is a flik), as well as an integer *size* field.

A map from $[0:i]$ to Kleene can be implemented using a single leaf for any $i < l - 1$. For $i \geq l$, we require a branch flik. A branch of size s implements a map from $[0:s-1]$ to Kleene by splitting the interval $[0:s-1]$ into a equal subintervals, each managed by a corresponding child. A subinterval where all the Kleene values are 0 need not be represented by a corresponding child (i.e., the child's value will be null). Further, a branch may be replaced by its first child if all of its other children are null.

A lookup or an update operation can be done with this representation in $O(\log N)$ time, where N is the size of the domain of the map. Note that the implementation is functional: an update returns a new tree and does not modify the original tree. As with all functional data structures, the new tree will share the unmodified parts of the old tree with the old tree.

We adapt the above data structure to represent maps from integers to an arbitrary set, by changing the representation of a leaf to store upto some fixed number l' of object references. We refer to this modified data structure as an *intmap*. An intmap can also be adapted to represent a map from *pairs of integers* to some arbitrary set, by first utilizing any suitable encoding function that maps every pair of integers to a unique integer. We refer to such an adaptation as an *intpairmap*. (Our implementation uses an encoding function that maps a pair (i, j) to the integer $(i + j) * (i + j + 1)/2 + i$, but other encoding schemes are possible.)

The values of unary predicates in a structure is represented using a *two-level map*: this consists of an intmap (the first-level map) that maps every node i in the structure's universe to a flik (the second-level map) that maps every unary predicate p's number to the value of $p(i)$ in the universe. The values of binary predicates is also represented by a two-level map: the first-level map (an intpairmap) maps a pair of individuals (m, n) to a second-level map (a flik) that maps a binary predicate p's number to the value of $p(m, n)$ in the structure.

[1] The data structure may be logically viewed as a tree, though sharing of subtrees can lead the actual representation to be a dag, just as in an OBDD.

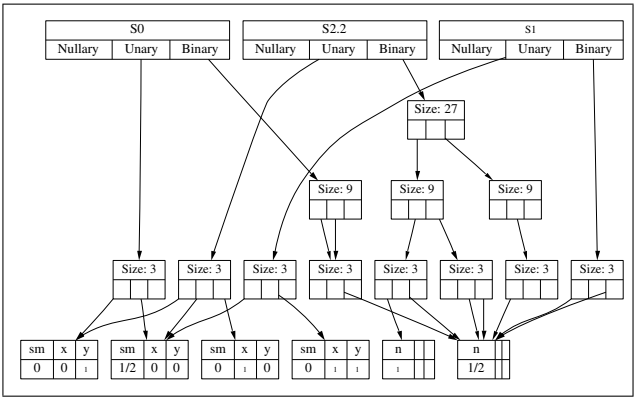


Fig. 6. The functional representation of the 3-valued structures in Fig. 3. Lists representing the universes are not shown in the figure. Nodes u_1 and u of structure S_0 are assigned numbers 0 and 1, respectively. Nodes u_1 and u of structure S_1 are, however, assigned numbers 1 and 0 respectively. Normalization causes this node number reassignment because the canonic name of node u_1 has changed. Nodes u_1, $u.0$, and $u.1$ of structure $S_{2.2}$ are assigned numbers 0, 1, and 2 respectively. The lowermost layer represents the second-level maps, while the layers above represent the first-level maps.

Fig. 6 shows a representation of the structures S_0, S_1 and $S_{2.2}$ from the running example. This representation uses a value of 3 for all the parameters a, l, and l'. Notice that even though there are a total of 7 nodes in the 3 structures, the representation requires only 4 fliks to represent their canonic names (i.e., the value of all unary predicates for these nodes).

Since the underlying implementation is functional, a 3-valued structure can be copied using a *shallow* copy of the pointers to the data structures described above. The resulting copy completely shares the underlying representation with the original. If this copy is subsequently modified via some sequence of operations, it will continue to share unchanged "inherited" portion of the representation.

Representation of Immutable TVS. Our implementation also utilizes *normalization*, which replaces distinct occurrences of equivalent data (which may be a fragment of the representation) by a unique or *canonical* representative. Normalization has two benefits: it increases the sharing between the representations of different structures and reduces the space requirements; it also makes it possible to check for equality (isomorphism) between structures more efficiently, e.g. through a pointer equality comparison. We normalize structures when they become immutable.

Normalization is done by first sorting nodes, based on their canonic names, and renumbering them from 0 on, and by then applying *hash-consing*: a hash table is used to store canonical representatives; a structured object *obj* is nor-

malized by first recursively normalizing its substructures and by then checking for the occurrence in the hash table of any object equivalent to *obj*; if such an object exists, *obj* is replaced by that object; otherwise, *obj* is added to the hash table.

5 Empirical Evaluation

In this section we present an empirical evaluation of the representations described in Section 4. Though we present timing results as well, our focus is primarily on evaluating the space requirements of the different representations. The experiments were conducted using TVLA version 0.92, running with SUN JDK 1.3, on a 1 GHZ Intel Pentium Processor machine with 1 GB RAM. The OBDD representation was implemented with the CUDD [15] package.

Benchmarks. The benchmarks used in the experiments are explained below. The `CA` benchmark performs "cleanness analysis" [4] in a C program implementing the instruction selection phase of the Tiger educational compiler [1]. The `GC` benchmark [12] involves a partial verification of the mark phase of a mark-and-sweep garbage collector. The `JFE` and the `Kernel` benchmarks are both instances of the Concurrent Modification Problem (CMP) described in [11]. CMP requires identifying a specific type of misuse of Java Collection Classes. Finally, the `MA` benchmark [10] verifies certain safety properties of a packet router in the mobile ambient calculus [3].

Results. Table 1 presents actual time and space statistics for running TVLA on the benchmark programs using each of our three implementations. These results indicate that both the OBDD implementation and the Functional implementations consume significantly less memory than the Base implementation. In the case of the OBDD implementation the table also shows how much memory was used by the CUDD package (which measures the memory used by the immutable structures that are represented as OBDDs) which is significantly less than the total memory used by the system, which additionally includes the memory used by the mutable structures in the Base representation. This shows the potential space reduction possible using a pure OBDD representation. The timing results show that the three implementations are comparable in performance.

Measuring Space Usage Via Instrumentation. Several factors confound a comparison of the different representations by measuring actual memory usage. In particular, actual memory usage is affected by the factors such as the programming language used, the runtime system used (e.g., a 16 byte object overhead in some Java implementations), libraries used, and the extent to which the implementation was carefully engineered. Hence, the actual memory usage is not a very accurate indicator of the memory that a representation actually requires.

Table 1. Time and space consumption of the representations. Time is measured in seconds and space in megabytes. The **Struct.** column denotes the number of structures. Recall that the OBDD representation is a phase-sensitive representation that uses the Base representation for mutable structures and OBDDs for immutable structures. In this case, we show both the memory used for representing all structures (the column labelled **Total Space**) and the memory used for representing immutable structures (the column labelled **Immutable Space**).

Benchmark	Struct.	Base		OBDD			Func.	
		Time	Space	Time	Total Space	Immutable Space	Time	Space
CA	40,000	1,861	168.2	1,874	32.6	12.6	4,567	12.9
GC	189,772	3,822	402.8	2,686	192.1	16.5	2,446	51.6
JFE	10,424	27	12.8	28	5.8	1.1	13	5.5
Kernel	6,079	29	22.7	40.2	6.8	3.4	22.4	16.7
MA	20,000	3,724	187.7	3,758	109	8.6	4,489	9.6

Therefore, we also instrumented our implementations to compute the actual number of objects of different type used by the different representations. From these counts we estimated the memory that would be required by a reasonable implementation, not counting overheads imposed by different runtime systems. Table 2 presents the statistics produced by our instrumentation. The table also reports two "metrics" (dense and sparse), which are different measures of the amount of information contained in the actual set of structures produced by the analysis. The *dense* metric is the space required to represent the structures, storing every predicate value explicitly in a bit-packed fashion, using two bits per predicate value. The *sparse* metric is the space required to store just the non-zero predicate values, using 4 bytes to identify each non-zero predicate value. The OBDD metric is obtained by multiplying the maximum number of live OBDD nodes that existed during the analysis (which is provided by the CUDD package) by 20, assuming that an OBDD node can be implemented using 20 bytes. The Functional metric is obtained by multiplying the number of objects used by the representation by 24 (as all objects in this implementation require 24 bytes or less). The Base representation metric is similarly computed, using appropriate sizes for the different kinds of objects used by this representation.

These results too indicate that our OBDD and Functional representations do very well and are better than the Base representation by an order of magnitude.

Asymptotic Trends. Since we are interested in a scalable TVS representation, we also measured how space consumption varies over the duration of the analysis. Specifically, we computed the instrumentation-based space usage estimate periodically. Table 3 shows that as the analysis proceeds, the average size of the structure increases, as measured by both the dense metric and the sparse metric. This is consistent with our expectations, since the state space exploration typically starts examining more complex structures, with more individuals, as

Table 2. Space counters for different representations. These counters indicate the number of bytes required to represent the structures. The **Struct.** column denotes the total number of structures produced by the analysis.

Benchmark	Struct.	Dense	Sparse	Base	OBDD	Func.
CA	40,000	7,473,737	11,053,352	22,516,937	2,302,140	1,749,384
GC	189,772	9,769,618	32,722,016	41,835,001	4,268,780	7,288,032
JFE	10,424	49,172,345	382,156	1,201,570	181,940	300,336
Kernel	6,079	64,768,292	799,604	2,292,436	420,520	315,168
MA	20,000	18,866,654	8,170,412	17,077,413	496,960	724,152

time proceeds. However, it may be seen that the space required to represent an average structure, decreases for the OBDD and Functional representation, indicating that the benefits of sharing increase as more structures are produced.

Table 3. Abstract counters used to represent TVS at different execution points of the iterative procedure. For MA we sample every $k = 2000$ structures, for CA $k = 5000$, and for GC $k = 10,000$. **D**, **S**, **B**, **O**, and **F** refer to the Dense, Sparse, Base, OBDD, and Functional representations respectively.

Sam-	MA					CA					GC				
ple	**D**	**S**	**B**	**O**	**F**	**D**	**S**	**B**	**O**	**F**	**D**	**S**	**B**	**O**	**F**
1	932	395	978	89	70	155	228	505	74	50	46	148	162	24	37
2	938	399	898	60	52	154	252	531	74	53	49	157	209	32	44
3	938	400	899	49	48	165	261	539	68	48	49	155	204	30	41
4	937	403	917	51	47	168	266	547	68	47	48	153	190	25	38
5	939	405	955	50	45	173	267	554	65	46	49	160	215	28	40
6	941	406	963	48	44	186	271	561	61	44	50	165	203	24	38
7	943	407	966	46	41	188	273	565	60	44	51	171	193	21	36
8	943	408	921	40	36	187	276	563	58	44	51	171	193	23	36
9	943	408	884	36	32	190	278	561	56	43	51	173	211	25	38
10	943	409	854	32	29	192	281	564	54	42	52	174	225	26	40

6 Related Work

Data Structures for Static Analysis Several recent and ongoing research efforts have explored the use of OBDDs in the context of static analysis, but mostly for domains simpler than first-order structures. PAG [7] and SLAM [9] use OBDDs to represent sets ("bit vectors") and interpretations of *propositional* (rather than first-order) structures. PAG also employs persistent data structures for static analysis, exploiting inherited sharing. Mona uses BDDs to represent

transitions of a tree automaton [5], which is used to implement a decision procedure used for Hoare-style verification. Mauborgne [8] explores the use of TDGs (a refinement of OBDDs) in abstract interpretation, using them to encode higher-order functions for strictness analysis, and presents empirical results on analysis time (but not on memory usage). It is not obvious from this prior work how OBDDs can be used beneficially for sophisticated analyses, such as heap shape analysis, that use domains based on first-order structures.

Using OBDDs to represent first order logical structures. Veith [16] describes a representation of a logical structure, where each predicate interpretation is encoded by a separate OBDD, and a vector of predicate interpretations is used to represent a structure. We achieve better sharing by encoding a complete structure using a single OBDD.

Other first-order state representations. Finally, we note that the composite symbolic library [19] can model a limited form of first-order state using Presburger formulas.

7 Conclusion and Future Work

In this paper, we have looked at efficient data structures for representing first-order structures in the context of software verification and sophisticated static analyses. We have described two new representations for first-order structures and have empirically measured the effectiveness of these representations. Our results show that these representations are very promising and that they can greatly reduce the space requirements for software verification. Our experience has been that these representations also help reduce the *time* required for software verification, though we have not addressed that aspect in detail in this paper. We are currently working on more efficient algorithms for the various high-level operations, such as *coerce* and *focus*, used in verification via 3-valued logic and hope to empirically evaluate the time complexity of these algorithms with the new representations described in this paper.

The representations described in this paper are for single 3-valued structures. A relational analysis, however, computes a *set* of 3-valued structures at each program point. We are also working on extensions to these representations that can represent sets of 3-valued structures (e.g., the BDD representation of single structures can be extended to represent a set of structures). It is worth noting that the empirical results presented in this paper show that our representation uses just a few (at most 3) objects per structure per program point, on the average. This implies that the representation is fairly efficient and that room for further savings exist only if we move on to extensions such as the representation of *sets of structures*.

References

1. A. Appel and M. Ginsburg. *Modern Compiler Implementation in C.* Cambridge University Press, New York, Cambridge, 1998.

2. R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.
3. L. Cardelli and A. Gordon. Mobile ambients. In M. Nivat, editor, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, Mar. 1998.
4. N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *Proc. Static Analysis Symp.*, pages 115–134, July 2000.
5. N. Klarlund, A. Moller, and M. I. Schwartzbach. MONA implementation secrets. In *CIAA*, pages 182–194, 2000.
6. T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In J. Palsberg, editor, *Proc. Static Analysis Symp.*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer-Verlag, 2000. http://www.cs.tau.ac.il/∼ rumster/TVLA/.
7. F. Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
8. L. Mauborgne. Abstract interpretation using typed decision graphs. *Science of Computer Programming*, 31(1):91–112, May 1998.
9. Microsoft Research. The SLAM project. http://research.microsoft.com/slam/, 2001.
10. F. Nielson, H. Nielson, and M. Sagiv. A Kleene analysis of mobile ambients. In G. Smolka, editor, *Proc. of ESOP 2000*, volume 1782 of *LNCS*, pages 305–319. Springer, 2000.
11. G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 83–94, Berlin, June 2002.
12. T. Reps, M. Sagiv, and R. Wilhelm. Automatic verification of a simple mark and sweep garbabge collector. Presented in the 2001 University of Washington and Microsoft Research Summer Institute, Specifying and Checking Properties of Software, http://research.microsoft.com/specncheck/, 2001.
13. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 105–118, 1999.
14. S. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm. A logic-based approach to data flow analysis problems. *Acta Inf.*, 35(6):457–504, June 1998.
15. F. Somenzi. Colorado University Decision Diagram Package (CUDD). Department of Electrical and Computer Engineering, University of Colorado at Boulder, 1998. http://vlsi.colorado.edu/∼ fabio/CUDD/.
16. H. Veith. How to encode a logical structure by an OBDD. In *IEEE Conference on Computational Complexity*, pages 122–131, 1998.
17. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 27–40, 2001.
18. B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A performance study of BDD-based model checking. In *Proceedings of the Formal Methods on Computer-Aided Design*, pages 255–289, November 1998.
19. T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. Composite symbolic library. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 335–344. Springer-Verlag, April 2001.

Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library*

Roberto Bagnara[1], Elisa Ricci[1], Enea Zaffanella[1], and Patricia M. Hill[2]

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

Abstract. The domain of convex polyhedra is employed in several systems for the analysis and verification of hardware and software components. Current applications span imperative, functional and logic languages, synchronous languages and synchronization protocols, real-time and hybrid systems. Since the seminal work of P. Cousot and N. Halbwachs, convex polyhedra have thus played an important role in the formal methods community and several critical tasks rely on their software implementations. Despite this, existing libraries for the manipulation of convex polyhedra are still research prototypes and suffer from limitations that make their usage problematic, especially in critical applications. Furthermore, there is inadequate support for polyhedra that are not necessarily closed (NNC), i.e., polyhedra that are described by systems of constraints where strict inequalities are allowed to occur. This paper presents the Parma Polyhedra Library, a new, robust and complete implementation of NNC convex polyhedra, concentrating on the distinctive features of the library and on the novel theoretical underpinnings.

Dedicated to the memory of Hervé Le Verge

1 Introduction

Convex polyhedra are regions of some n-dimensional space that are bounded by a finite set of hyperplanes. A convex polyhedron in $\mathbb{R}^n$ describes a relation between n real-valued quantities. The seminal work of P. Cousot and N. Halbwachs [15] introduced the use of convex polyhedra as a domain of descriptions to solve, by *abstract interpretation* [13], a number of important data-flow analysis problems such as array bound checking, compile-time overflow detection, loop invariant computations and loop induction variables. Convex polyhedra are also used, among many other applications, for the analysis and verification of

* This work has been partly supported by MURST projects "Abstract Interpretation, type systems and control-flow analysis" and "Aggregate- and number-reasoning for computing: from decision algorithms to constraint programming with multisets, sets, and maps" and by EPSRC grant GR/R53401/01.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 213–229, 2002.
© Springer-Verlag Berlin Heidelberg 2002

synchronous languages [6,18] and of linear hybrid automata (an extension of finite-state machines that models time requirements) [20,22], for the computer-aided formal verification of concurrent and reactive systems based on temporal specifications [27], for inferring argument size relationships in logic languages [5], and for the automatic parallelization of imperative programs [30]. Since the work of Cousot and Halbwachs, convex polyhedra have thus played an important role in the formal methods community and new uses continue to emerge (see, e.g., [11, 16]). As a consequence, several critical tasks, such as checking the correctness of synchronization protocols or verifying the absence of run-time errors of systems whose failure can cause serious damage, rely on the software implementations of convex polyhedra.

Traditionally, convex polyhedra are assumed to be topologically closed and described by constraint systems containing linear equations and non-strict linear inequalities. However, some tasks need to use convex polyhedra that are not necessarily closed (NNC), i.e., polyhedra that are described by constraint systems possibly containing strict linear inequalities (in addition to equations and non-strict inequalities). Strict inequalities are important, for instance, in order to directly represent non-intersecting temporal regions, as is often the case when modeling applications where synchronization protocols, asynchronous interactions and temporal constraints come into play. Recently, they have also been used for the automatic computation of linear ranking functions [11].

Prior to the release of the Parma Polyhedra Library (PPL) [2] which is the subject of this paper, four libraries for the manipulation of convex polyhedra were (and continued to be) available:[1]

1. Polylib, designed and written by H. Le Verge and D. K. Wilde [25,32];
2. *PolyLib*, the successor of the library by Le Verge and Wilde [26];
3. New Polka, by B. Jeannet [24];
4. the polyhedra library that comes with the HyTech tool [22].

All libraries,[2] including the PPL, use the same basic technique for representing convex polyhedra (the *double description method* recalled in Section 3) and the same core algorithm for manipulating that representation (the implementation and extension of N. V. Chernikova's algorithms [8,9,10] by Le Verge [25], possibly with the improvements of K. Fukuda and A. Prodon [17]).

Apart from the PPL, only New Polka supports NNC polyhedra. However, this support is incomplete, incurring avoidable inefficiencies and leaving the client application with the non-trivial task of a correct interpretation of the results.

Libraries 1 and 4 may incur overflow problems whereas 2 and 3 can use unbounded integers as coefficients. Libraries 2 and 3 can be configured to use finite integral types for extra speed but this, of course, comes with the possibility of overflows.

[1] We restrict ourselves to those libraries that are freely available and provide the services required by applications in static analysis and computer-aided verification.

[2] We refer to the following versions that, at the time of writing, are the latest available: Polylib 2.1, *PolyLib* 5.0.4, New Polka 1.1.3c, HyTech 1.04f.

In libraries 1–4, matrices of coefficients, which are the main data structures used to represent polyhedra, cannot grow dynamically and the client application is ultimately responsible for specifying their dimensions. Since the worst case space complexity of the methods employed is exponential, in general the client application cannot make a safe and practical choice: specifying small dimensions may provoke a run-time failure; generous dimensions may waste significant amounts of memory and, again, result in unnecessary run-time failures.

The problems caused by run-time errors such as overflow and memory allocation failure could be mitigated or even solved by suitable mechanisms for error detection, handling and recovery. This requires the ability to detect the problem, releasing any affected data-structure whether it be completely or only partially constructed,and continue the computation with an alternative method (e.g., by reverting to an interval-based approximation). Library 1 detects some errors and sets an error flag whereas libraries 3 and 4 detect some errors, print an error message and abort. Library 2 detects more errors, sometimes setting a flag and sometimes printing a message and aborting.

Libraries 1–3 are free software released under the GNU General Public License (GPL, see `http://www.gnu.org/`). Thus, when faced with the need to overcome the above mentioned limitations, anyone can freely take and use them as the basis for further development. However, it appears that none of the libraries provide documentation for the interfaces and code that is adequate for an outsider to make such improvements with any real confidence. This feeling of insecurity is aggravated by the discovery of some errors and imprecisions in the theoretical sections of the documentation of some libraries (for more information see [3], which itself contains an error, now corrected in [4]). Moreover, a complete solution to issues such as error recovery and fully dynamic memory allocation requires a somewhat radical departure from the existing code bases.

For all these reasons we decided to write the PPL, a robust and complete implementation of convex polyhedra. This paper describes the library concentrating on some of its distinctive features and the novel theoretical underpinnings.

The plan of the paper is as follows: Section 2 recalls some basic notation and terminology; Section 3 introduces convex polyhedra and the double description method paying attention to common pitfalls; Section 4 presents the NNC polyhedra and describes (from a theoretical perspective) how they are handled in the PPL; Section 5 briefly describes the design and implementation of the PPL and how it addresses all the limitations we have just discussed; Section 6 concludes.

A longer version of this paper, containing the proofs of the results presented here, is available as a technical report [3].

2 Preliminaries

In this paper, all topological arguments refer to the topological space $\mathbb{R}^n$ with the standard topology. The *topological closure* of $S \subseteq \mathbb{R}^n$ is denoted by $\mathbb{C}(S)$ and defined as $\mathbb{C}(S) \stackrel{\text{def}}{=} \bigcap\{\, C \subseteq \mathbb{R}^n \mid S \subseteq C \text{ and } C \text{ is closed}\,\}$. We denote the set of all non-negative reals by $\mathbb{R}_+$. For each $i \in \{1, \ldots, n\}$, v_i denotes the i-th component of the (column) vector $\boldsymbol{v} \in \mathbb{R}^n$. We denote by $\mathbf{0}$ the vector of $\mathbb{R}^n$,

called *the origin*, having all components equal to zero. A vector $\boldsymbol{v} \in \mathbb{R}^n$ can also be interpreted as a matrix in $\mathbb{R}^{n\times 1}$ and manipulated accordingly using the usual definitions for addition, multiplication (both by a scalar and by another matrix), and transposition, which is denoted by $\boldsymbol{v}^{\mathrm{T}}$. The *scalar product* of $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$, denoted $\langle \boldsymbol{v}, \boldsymbol{w} \rangle$, is the real number $\boldsymbol{v}^{\mathrm{T}}\boldsymbol{w} = \sum_{i=1}^{n} v_i w_i$. For any relational operator $\bowtie \in \{=, \geq, \leq, <, >\}$, we write $\boldsymbol{v} \bowtie \boldsymbol{w}$ to denote the conjunctive proposition $\bigwedge_{i=1}^{n} (v_i \bowtie w_i)$. In contrast, $\boldsymbol{v} \neq \boldsymbol{w}$ will denote the proposition $\neg(\boldsymbol{v} = \boldsymbol{w})$. We will sometimes use the convenient notation $a \bowtie_1 b \bowtie_2 c$ to denote the conjunction $a \bowtie_1 b \wedge b \bowtie_2 c$ and we will not distinguish conjunctions of propositions from sets of propositions. For each set $S \subseteq \mathbb{R}^n$ of finite cardinality m, we denote by $\mathrm{matrix}(S) \subseteq \mathbb{R}^{n\times m}$ the set of all matrices having S as the set of their columns.

3 Convex Polyhedra and the Double Description Method

For each vector $\boldsymbol{a} \in \mathbb{R}^n$ and scalar $b \in \mathbb{R}$, where $\boldsymbol{a} \neq \boldsymbol{0}$, the linear (non-strict) inequality constraint $\langle \boldsymbol{a}, \boldsymbol{x} \rangle \geq b$ defines a closed affine half-space. The linear equality constraint $\langle \boldsymbol{a}, \boldsymbol{x} \rangle = b$ defines an affine hyperplane. Convex polyhedra are usually described as finite systems of linear equality and inequality constraints. When working at the theoretical level, it is simpler to express each equality constraint as the intersection of the two half-spaces $\langle \boldsymbol{a}, \boldsymbol{x} \rangle \geq b$ and $\langle -\boldsymbol{a}, \boldsymbol{x} \rangle \geq -b$.

Definition 1. (Closed polyhedron.) *The set $\mathcal{P} \subseteq \mathbb{R}^n$ is a* closed polyhedron *if and only if either $\mathcal{P}$ can be expressed as the intersection of a finite number of closed affine half-spaces of $\mathbb{R}^n$, or $n = 0$ and $\mathcal{P} = \varnothing$.*

Alternatively, the definition of a convex polyhedron can be based on some of the geometric features of the set of solutions of such a system of constraints. A vector $\boldsymbol{r} \in \mathbb{R}^n$ such that $\boldsymbol{r} \neq \boldsymbol{0}$ is a *ray* (or *direction of infinity*) of a non-empty polyhedron $\mathcal{P} \subseteq \mathbb{R}^n$ if, for every point $\boldsymbol{p} \in \mathcal{P}$ and every $\lambda \in \mathbb{R}_+$, it holds $\boldsymbol{p} + \lambda \boldsymbol{r} \in \mathcal{P}$; a vector $\boldsymbol{l} \in \mathbb{R}^n$ is a *line* of $\mathcal{P}$ if both $\boldsymbol{l}$ and $-\boldsymbol{l}$ are rays of $\mathcal{P}$. The empty polyhedron has no rays and no lines. As was the case for equality constraints, the theory can dispense with the use of lines by using the corresponding pair of opposite rays. The following theorem is a simple consequence of well known theorems by Minkowski and Weyl [31].

Theorem 1. *The set $\mathcal{P} \subseteq \mathbb{R}^n$ is a closed polyhedron if and only if there exist finite sets $R, P \subseteq \mathbb{R}^n$ of cardinality k and ℓ, respectively, such that $\boldsymbol{0} \notin R$ and, for any matrices $K \in \mathbb{R}^{n\times k}$ and $L \in \mathbb{R}^{n\times \ell}$ where $K \in \mathrm{matrix}(R)$ and $L \in \mathrm{matrix}(P)$,*

$$\mathcal{P} = \left\{ K\boldsymbol{\mu} + L\boldsymbol{\nu} \in \mathbb{R}^n \;\middle|\; \boldsymbol{\mu} \in \mathbb{R}_+^k, \boldsymbol{\nu} \in \mathbb{R}_+^\ell, \textstyle\sum_{i=1}^{\ell} \nu_i = 1 \right\}.$$

When $\mathcal{P} \neq \varnothing$, we say that $\mathcal{P}$ is described by the *generator system* $\mathcal{G} = (R, P)$. In particular, the vectors of R and P are rays and points of $\mathcal{P}$, respectively. Informally speaking, if no "supporting point" is provided then an empty polyhedron is obtained; formally, $\mathcal{P} = \varnothing$ if and only if $P = \varnothing$. By convention, the empty system (i.e., the system with $R = \varnothing$ and $P = \varnothing$) is the only generator system

for the empty polyhedron. It is worth stressing that, in general, the vectors in R and P are not the *extreme rays* and the *vertices* of the polyhedron [3]: for instance, any half-space of $\mathbb{R}^2$ has two extreme rays and no vertices, but any generator system describing it will contain at least three rays and one point.

The combination of the two approaches outlined above is the basis of the *Double Description* (DD) method [29], which exploits the duality principle to compute each representation starting from the other one, possibly minimizing the descriptions. We will write $\mathrm{con}(\mathcal{C})$ and $\mathrm{gen}(\mathcal{G})$ to denote the polyhedra described by the finite constraint system $\mathcal{C}$ and generator system $\mathcal{G}$, respectively.

Definition 2. (DD pair and minimal forms.) *If* $\mathrm{con}(\mathcal{C}) = \mathrm{gen}(\mathcal{G}) = \mathcal{P}$, *then* $(\mathcal{C}, \mathcal{G})$ *is said to be a* DD pair *for* $\mathcal{P}$, *and we write* $(\mathcal{C}, \mathcal{G}) \equiv \mathcal{P}$. *We say that*

- $\mathcal{C}$ *is in* minimal form *if there does not exist* $\mathcal{C}' \subset \mathcal{C}$ *such that* $\mathrm{con}(\mathcal{C}') = \mathcal{P}$;
- $\mathcal{G} = (R, P)$ *is in* minimal form *if there does not exist* $\mathcal{G}' = (R', P')$ *such that* $\mathcal{G}' \neq \mathcal{G}$, $R' \subseteq R$, $P' \subseteq P$ *and* $\mathrm{gen}(\mathcal{G}') = \mathcal{P}$;
- *the DD pair* $(\mathcal{C}, \mathcal{G})$ *is in* minimal form *if* $\mathcal{C}$ *and* $\mathcal{G}$ *are both in minimal form.*

The set of all closed polyhedra on the vector space $\mathbb{R}^n$, denoted $\mathbb{CP}_n$, can be partially ordered by set-inclusion to form a lattice having the empty set and $\mathbb{R}^n$ as the bottom and top element, respectively. The binary meet operation is thus given by set-intersection, which is easily implemented by taking the union of the constraint systems representing the two arguments. The binary join operation, denoted $\uplus$ and called *convex polyhedral hull* (poly-hull, for short), is implemented by taking the union of the two arguments' generator systems. Note that, in general, the poly-hull of two polyhedra is different from their convex hull [31].

4 Handling Not Necessarily Closed Polyhedra

For each vector $\boldsymbol{a} \in \mathbb{R}^n$ and scalar $b \in \mathbb{R}$, where $\boldsymbol{a} \neq \boldsymbol{0}$, the linear strict inequality constraint $\langle \boldsymbol{a}, \boldsymbol{x} \rangle > b$ defines an open affine half-space. By allowing strict inequalities to occur in the system of constraints, it is possible to define polyhedra that are not necessarily closed (NNC polyhedra, for short).

Definition 3. (NNC polyhedron.) *The set* $\mathcal{P} \subseteq \mathbb{R}^n$ *is a* NNC polyhedron *if and only if either* $\mathcal{P}$ *can be expressed as the intersection of a finite number of (not necessarily closed) affine half-spaces of* $\mathbb{R}^n$, *or* $n = 0$ *and* $\mathcal{P} = \varnothing$.

We denote by $\mathbb{P}_n$ the set of all NNC polyhedra on the vector space $\mathbb{R}^n$. Obviously, we have $\mathbb{CP}_n \subseteq \mathbb{P}_n$ (note that $\mathbb{CP}_n = \mathbb{P}_n$ if and only if $n = 0$). When partially ordered by set-inclusion, $\mathbb{P}_n$ is a lattice and $\mathbb{CP}_n$ is a sublattice of $\mathbb{P}_n$.

To the best of the authors' knowledge, the first software library (based on the DD method) allowing for the computation over the domain $\mathbb{P}_n$ was the Polka library [20], where each NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$ is embedded into a closed polyhedron $\mathcal{R} \in \mathbb{CP}_{n+1}$. The additional dimension of the vector space, usually labeled by the letter ϵ, encodes the topological closeness of each affine half-space in the constraint description for $\mathcal{P}$. Namely, if $\mathcal{P} = \mathrm{con}(\mathcal{C})$, where

$$\mathcal{C} = \big\{ \langle \boldsymbol{a}_i, \boldsymbol{x} \rangle \bowtie_i b_i \;\big|\; i \in \{1, \ldots, m\}, \boldsymbol{a}_i \in \mathbb{R}^n, \bowtie_i \in \{\geq, >\}, b_i \in \mathbb{R} \big\},$$

then the representation polyhedron is defined as $\mathcal{R} = \mathrm{con}\big(\mathrm{con\_repr}(\mathcal{C})\big)$, where

$$\begin{aligned}\mathrm{con\_repr}(\mathcal{C}) \stackrel{\mathrm{def}}{=} \bigl\{0 \le \epsilon \le 1\bigr\} \\ \cup \bigl\{\, \langle \boldsymbol{a}_i, \boldsymbol{x}\rangle - 1 \cdot \epsilon \ge b_i \;\bigm|\; i \in \{1, \dots, m\}, \bowtie_i \in \{>\} \,\bigr\} \\ \cup \bigl\{\, \langle \boldsymbol{a}_i, \boldsymbol{x}\rangle + 0 \cdot \epsilon \ge b_i \;\bigm|\; i \in \{1, \dots, m\}, \bowtie_i \in \{\ge\} \,\bigr\}.\end{aligned}$$

It should be stressed that the choice of the value -1 for the coefficients of the additional variable ϵ in the constraints representing strict inequalities is rather arbitrary: any other negative value will do. Similarly, the side constraint $\epsilon \le 1$ could be replaced by any other *ϵ-upper-bound* constraint, i.e., by any constraint $\epsilon \le \delta$ such that $\delta > 0$. Different, though equivalent, constraint systems $\mathcal{C}_j$ describing $\mathcal{P}$ may be embedded into different representation polyhedra $\mathcal{R}_j = \mathrm{con}\bigl(\mathrm{con\_repr}(\mathcal{C}_j)\bigr)$. We shall abuse notation by writing $\mathcal{R} = \mathrm{con\_repr}(\mathcal{P})$ as a shorthand for $\mathcal{R} = \mathrm{con}\bigl(\mathrm{con\_repr}(\mathcal{C})\bigr)$, provided the constraint system $\mathcal{C}$ describing $\mathcal{P}$ is clear from context.

4.1 The Generators of NNC Polyhedra

One of the fundamental features of the DD method, and the very reason for its name, is the ability to represent a closed polyhedron using a system of constraints or a system of generators. While being equivalent, there are contexts where each of these descriptions is the most appropriate, so that a good library should provide the client application with both possibilities.

Any NNC polyhedron can be easily described by using constraint systems containing strict inequalities, but a similar generalization of the concept of generator system seems to be missing. This causes an asymmetry in the handling of NNC polyhedra using the DD method that is reflected in existing software libraries. For instance, the following sentence comes from the documentation of New Polka [24, Section 1.1.4, page 10] (where s denotes the ϵ coefficient):

> Don't ask me the intuitive meaning of s $/= 0$ in rays and vertices !

The problem is discussed in some more detail in [19, Section 4.5, pp. 10–11]:

> While strict inequations handling is transparent for constraints (being displayed accurately), the extra dimension added to the variables space is apparent when it comes to generators : one extra coefficient, resp. extra vertices (as `epsilon` is bounded), materialize this dimension in every generator, resp. generators system.
> This makes more difficult to define polyhedra with the only help of generators : one should carefully study the extra vertices with non null `epsilon` coefficients added to constraints defined polyhedra, in the case of large inequations, and the case of strict inequations.

A clear understanding of the generator systems of NNC polyhedra is helpful, for instance, in order to generalize to the domain $\mathbb{P}_n$ those operators on closed polyhedra that are defined or implemented in terms of the generator system representation (such as the *time-elapse* operator of [20,21] or the generators-based widening of [6]).

We will now show how, by decoupling the user interface from the details of the particular implementation, it is possible to provide an intuitive generalization of the concept of generator system, so that the geometric features of any NNC polyhedron can be accurately represented. The key step is the introduction of a new kind of generator.

Definition 4. (Closure point.) *A vector* $\boldsymbol{c} \in \mathbb{R}^n$ *is a* closure point *of* $S \subseteq \mathbb{R}^n$ *if and only if* $\boldsymbol{c} \in \mathbb{C}(S)$.

When considering NNC polyhedra, closure points can be characterized by a property which is similar to the one used when defining rays.

Proposition 1. *A vector* $\boldsymbol{c} \in \mathbb{R}^n$ *is a closure point of the NNC polyhedron* $\mathcal{P} \in \mathbb{P}_n$ *if and only if* $\mathcal{P} \neq \varnothing$ *and* $\lambda\boldsymbol{p} + (1-\lambda)\boldsymbol{c} \in \mathcal{P}$ *for every point* $\boldsymbol{p} \in \mathcal{P}$ *and* $\lambda \in \mathbb{R}$ *such that* $0 < \lambda < 1$.

We are now able to provide a parametric description for any NNC polyhedron.

Theorem 2. *The set* $\mathcal{P} \subseteq \mathbb{R}^n$ *is an NNC polyhedron if and only if there exist finite sets* $R, P, C \subseteq \mathbb{R}^n$ *of cardinality* k, ℓ *and* m, *respectively, such that* $\mathbf{0} \notin R$ *and, for any matrices* $K \in \mathbb{R}^{n\times k}$, $L \in \mathbb{R}^{n\times \ell}$, $M \in \mathbb{R}^{n\times m}$ *where* $K \in \mathrm{matrix}(R)$, $L \in \mathrm{matrix}(P)$ *and* $M \in \mathrm{matrix}(C)$,

$$\mathcal{P} = \left\{ K\boldsymbol{\mu} + L\boldsymbol{\nu} + M\boldsymbol{\eta} \in \mathbb{R}^n \;\middle|\; \begin{array}{l} \boldsymbol{\mu} \in \mathbb{R}^k_+, \boldsymbol{\nu} \in \mathbb{R}^\ell_+, \boldsymbol{\nu} \neq \boldsymbol{\theta}, \boldsymbol{\eta} \in \mathbb{R}^m_+, \\ \sum_{i=1}^{\ell} \nu_i + \sum_{i=1}^{m} \eta_i = 1 \end{array} \right\}.$$

When $\mathcal{P} \neq \varnothing$, we say that $\mathcal{P}$ is described by the *extended generator system* $\mathcal{G} = (R, P, C)$. As was the case for closed polyhedra, the vectors in R and P are rays and points of $\mathcal{P}$, respectively. The condition $\boldsymbol{\nu} \neq \boldsymbol{\theta}$ ensures that at least one of the points of P plays an active role in any convex combination of the vectors of P and C. It follows from Proposition 1 that the vectors of C are closure points of $\mathcal{P}$. Since rays and closure points need a supporting point, we have $\mathcal{P} = \varnothing$ if and only if $P = \varnothing$.

In Figure 1, we provide a few examples of the use of extended generator systems for the description of NNC polyhedra: (closure) points are represented by small (un-) filled circles, whereas rays are represented by vectors that, for notational convenience, are applied to points. The NNC polyhedron $\mathcal{P}_1$ is an

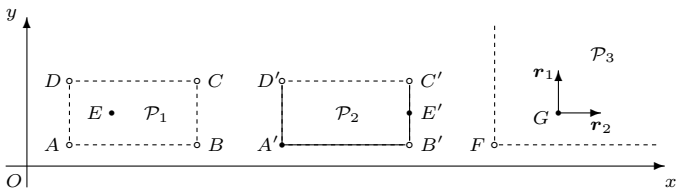


Fig. 1. Using closure points to define NNC polyhedra.

open rectangle and is described by the closure points A, B, C, D and the point E; note that E could have been replaced by any other point of $\mathcal{P}_1$, whereas all the four closure points have to be included in any generator system for $\mathcal{P}_1$. The NNC polyhedron $\mathcal{P}_2$ is another rectangle that is neither closed nor open: since A' is a point, the open segments $]A', B'[$ and $]A', D'[$ are included in $\mathcal{P}_2$; similarly, the open segment $]B', C'[$ is included in $\mathcal{P}_2$ because E' is a point of the generator system (note that E' is needed, since both B' and C' are not in $\mathcal{P}_2$, but it could have been replaced by any other point lying on this open segment); in contrast, the closed segment $[C', D']$ is disjoint from $\mathcal{P}_2$, because neither C' nor D' are points of $\mathcal{P}_2$. Finally, the NNC polyhedron $\mathcal{P}_3$ can be regarded as the translation by F of the open positive quadrant. Thus the generator system includes the closure point F, the rays $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ and the point G; again, the latter could have been replaced by any other point of $\mathcal{P}_3$.

We will now show how the high level description of an NNC polyhedron provided by an extended generator system can be mapped into an implementation based on the ϵ dimension approach. Namely, if $\mathcal{G} = (R, P, C)$ is the extended generator system describing $\mathcal{P} \in \mathbb{P}_n$, the corresponding closed representation $\mathcal{R} \in \mathbb{CP}_{n+1}$ is described by the generator system $\mathrm{gen\_repr}(\mathcal{G}) \stackrel{\mathrm{def}}{=} (R', P')$ where

$$\begin{aligned} R' &= \bigl\{\, (\boldsymbol{r}^{\mathrm{T}}, 0)^{\mathrm{T}} \mid \boldsymbol{r} \in R \,\bigr\}, \\ P' &= \bigl\{\, (\boldsymbol{p}^{\mathrm{T}}, 1)^{\mathrm{T}} \mid \boldsymbol{p} \in P \,\bigr\} \cup \bigl\{\, (\boldsymbol{x}^{\mathrm{T}}, 0)^{\mathrm{T}} \mid \boldsymbol{x} \in P \cup C \,\bigr\}. \end{aligned}$$

Even in this case, the value 1 for the coordinate of the ϵ dimension in the translation of points is almost arbitrary: any other positive value could be chosen. Different though equivalent extended generator systems $\mathcal{G}_j$ describing $\mathcal{P} \in \mathbb{P}_n$ may result in different representation polyhedra $\mathcal{R}_j = \mathrm{gen}\bigl(\mathrm{gen\_repr}(\mathcal{G}_j)\bigr)$. It is worth noting that the closure points of $\mathcal{P}$ are mapped to points of $\mathcal{R}$ having a zero ϵ coordinate. In contrast, the points of $\mathcal{P}$ are mapped to a pair of points of $\mathcal{R}$, having a zero and a strictly positive ϵ coordinate, respectively; by doing this, we explicitly enforce in the closed representation the key invariant saying that any point of $\mathcal{P}$ is also a closure point of $\mathcal{P}$.

Definition 5. (ϵ-representation.) *A polyhedron $\mathcal{R} \in \mathbb{CP}_{n+1}$ is said to be an* ϵ-representation *if and only if*

$$\exists \delta > 0 \,.\, \mathcal{R} \subseteq \mathrm{con}\bigl(\{0 \le \epsilon \le \delta\}\bigr); \tag{1}$$

$$\bigl(\exists \epsilon > 0 \,.\, (\boldsymbol{x}^{\mathrm{T}}, \epsilon)^{\mathrm{T}} \in \mathcal{R}\bigr) \implies (\boldsymbol{x}^{\mathrm{T}}, 0)^{\mathrm{T}} \in \mathcal{R}. \tag{2}$$

$\mathcal{R}$ is said to be an ϵ-representation for *$\mathcal{P} \in \mathbb{P}_n$, denoted $\mathcal{R} \Rrightarrow_\epsilon \mathcal{P}$, if $\mathcal{R}$ is an ϵ-representation and*

$$\mathcal{P} = [\![\mathcal{R}]\!] \stackrel{\mathrm{def}}{=} \bigl\{\, \boldsymbol{x} \in \mathbb{R}^n \mid \exists \epsilon > 0 \,.\, (\boldsymbol{x}^{\mathrm{T}}, \epsilon)^{\mathrm{T}} \in \mathcal{R} \,\bigr\}. \tag{3}$$

Proposition 2. *Let $(\mathcal{C}, \mathcal{G}) \equiv \mathcal{P} \in \mathbb{P}_n$. Then we have* $\mathrm{con}\bigl(\mathrm{con\_repr}(\mathcal{C})\bigr) \Rrightarrow_\epsilon \mathcal{P}$ *and* $\mathrm{gen}\bigl(\mathrm{gen\_repr}(\mathcal{G})\bigr) \Rrightarrow_\epsilon \mathcal{P}$.

Operations such as the intersection of NNC polyhedra and the application of affine transformations can be safely performed on any of the ϵ-representations of the arguments; the same holds for the poly-hull operation, provided none of the arguments is an empty NNC polyhedron. Some care has to be taken when testing the emptiness of an NNC polyhedron or the inclusion of an NNC polyhedron into another one [20]. For instance, any ϵ-representation included in the hyperplane $\epsilon = 0$ actually encodes the empty NNC polyhedron.

Proposition 3. *Let $\mathcal{R} \Rrightarrow_\epsilon \mathcal{P}$, $\mathcal{R}_1 \Rrightarrow_\epsilon \mathcal{P}_1$ and $\mathcal{R}_2 \Rrightarrow_\epsilon \mathcal{P}_2$. Then*

1. *$\mathcal{P} = \varnothing$ if and only if $\mathcal{R} \subseteq \mathrm{con}\bigl(\{\epsilon \leq 0\}\bigr)$;*
2. *$\mathcal{R}_1 \cap \mathcal{R}_2 \Rrightarrow_\epsilon \mathcal{P}_1 \cap \mathcal{P}_2$;*
3. *$(\mathcal{P}_1 \not\varnothing \wedge \mathcal{P}_2 \not\varnothing) \implies (\mathcal{R}_1 \uplus \mathcal{R}_2 \Rrightarrow_\epsilon \mathcal{P}_1 \uplus \mathcal{P}_2)$;*
4. *let $f \stackrel{\text{def}}{=} \lambda \boldsymbol{x} \in \mathbb{R}^n \,.\, A\boldsymbol{x} + \boldsymbol{b}$ be any affine transformation defined on $\mathbb{P}_n$; then $g(\mathcal{R}) \Rrightarrow_\epsilon f(\mathcal{P})$, where*

$$g \stackrel{\text{def}}{=} \lambda \begin{pmatrix} \boldsymbol{x} \\ \epsilon \end{pmatrix} \in \mathbb{R}^{n+1} \,.\, \begin{pmatrix} A & \boldsymbol{0} \\ \boldsymbol{0}^{\mathrm{T}} & 1 \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \epsilon \end{pmatrix} + \begin{pmatrix} \boldsymbol{b} \\ 0 \end{pmatrix}$$

is the corresponding affine transformation on $\mathbb{CP}_{n+1}$.

4.2 The Issue of Minimization

When adopting the ϵ dimension approach proposed in [20], the computed representation $\mathcal{R} \in \mathbb{CP}_{n+1}$ of an NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$ will depend not only on the particular constraint system considered, but also on the sequence of operations (intersections, poly-hulls, affine transformations, etc.) performed on the polyhedron. If not properly handled, such an abundance of possible representations may cause problems when trying to provide a non-redundant description of $\mathcal{P}$.

The reason is that libraries such as New Polka compute the minimal forms of the closed representation $\mathcal{R}$. Very often, such an approach results in a redundant description of the represented NNC polyhedron: there may be a different ϵ-representation for $\mathcal{P}$ that is characterized by a smaller number of constraints (generators). The following example illustrates this point.

Consider the two NNC polyhedra $\mathcal{P}_1, \mathcal{P}_2 \in \mathbb{P}_1$ defined as

$$\mathcal{P}_1 \stackrel{\text{def}}{=} \mathrm{con}\bigl(\{0 < x < 2\}\bigr), \qquad \mathcal{P}_2 \stackrel{\text{def}}{=} \mathrm{con}\bigl(\{2 < x < 3\}\bigr).$$

These polyhedra are encoded by the closed polyhedra $\mathcal{R}_1, \mathcal{R}_2 \in \mathbb{CP}_2$ such that[3]

$$\mathcal{R}_1 \stackrel{\text{def}}{=} \mathrm{con\_repr}(\mathcal{P}_1) = \bigl\{\, (x, \epsilon)^{\mathrm{T}} \in \mathbb{R}^2 \;\bigm|\; \epsilon \geq 0,\ x - \epsilon \geq 0,\ -x - \epsilon \geq -2 \,\bigr\},$$
$$\mathcal{R}_2 \stackrel{\text{def}}{=} \mathrm{con\_repr}(\mathcal{P}_2) = \bigl\{\, (x, \epsilon)^{\mathrm{T}} \in \mathbb{R}^2 \;\bigm|\; \epsilon \geq 0,\ x - \epsilon \geq 2,\ -x - \epsilon \geq -3 \bigr\}.$$

Suppose now that the user wants to compute the poly-hull of the two original polyhedra, therefore obtaining the NNC polyhedron $\mathcal{P}_3 \stackrel{\text{def}}{=} \mathrm{con}\bigl(\{0 < x < 3\}\bigr)$.

[3] In both cases, we do not explicitly include the ϵ-upper-bound constraint $\epsilon \leq 1$, which happens to be redundant.

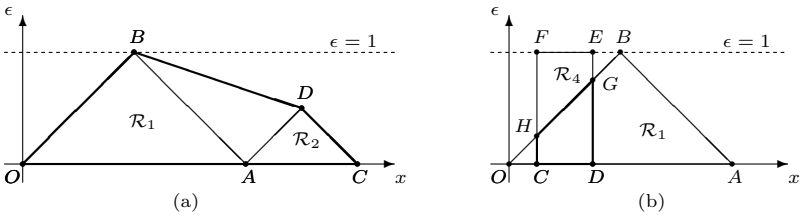


Fig. 2. (a) The poly-hull of $\mathcal{R}_1$ and $\mathcal{R}_2$; (b) The intersection of $\mathcal{R}_1$ and $\mathcal{R}_4$.

At the representation level, the situation will be the one described in Figure 2(a): $\mathcal{P}_3$ is represented by the closed polyhedron generated by the four vertices O, C, D, and B, whereas point A is identified as redundant. Formally,

$$\mathcal{R}_3 \stackrel{\text{def}}{=} \bigl\{\, (x,\epsilon)^{\mathrm{T}} \in \mathbb{R}^2 \;\bigm|\; \epsilon \geq 0,\ x-\epsilon \geq 0,\ -x-\epsilon \geq -3,\ -x-3\epsilon \geq -4 \,\bigr\}.$$

The last non-strict inequality, which corresponds to the segment $[B, D]$ in Figure 2(a), is not redundant as far as the ϵ-representation $\mathcal{R}_3$ is concerned. However, this non-strict inequality stands for the strict inequality $x < 4$, which is clearly redundant when considering the represented polyhedron $\mathcal{P}_3$.

The problem outlined above is even more critical when dealing with higher dimension vector spaces: it is straightforward to devise examples where more than half of the constraints in the "minimized" representation happen to be redundant. Even when disregarding these pathological cases, redundancy could have a serious negative impact on the efficiency of some of the operations computed on the polyhedron, e.g., those characterized by a worst case complexity that is exponential in the size of the description.

Besides efficiency issues, the presence of redundant constraints may also cause headaches to the users of the library. For instance, suppose one wants to know if a given NNC polyhedron is not topologically closed. Ordinary users (i.e., all the users but the experts) may be tempted to implement such a test by checking whether the constraint system in minimal form contains any strict inequality. Unfortunately, such an approach would be unsound, as can be easily observed by considering the scenario proposed in Figure 2(b). Here, the NNC polyhedron $\mathcal{P}_1$ is intersected with the NNC polyhedron $\mathcal{P}_4 \stackrel{\text{def}}{=} \mathrm{con}\bigl(\{1 \leq 4x \leq 3\}\bigr)$, whose representation $\mathcal{R}_4 = \mathrm{con\_repr}(\mathcal{P}_4)$ is the rectangle having vertices C, D, E and F. The resulting trapezium is another ϵ-representation for the NNC polyhedron $\mathcal{P}_4$, which is topologically closed. However, any constraint system describing the trapezium will also encode the strict inequality $x > 0$, corresponding to the closed segment $[G, H]$.

It is therefore meaningful to address the problem of providing a minimization procedure that is able to remove all of these redundancies. To this end, the introduction of some notation will be helpful.

We say that a vector $\boldsymbol{v}$ *saturates* the constraint $\langle \boldsymbol{a}, \boldsymbol{x} \rangle \bowtie b$ if and only if $\langle \boldsymbol{a}, \boldsymbol{v} \rangle = b$. For any constraint system $\mathcal{C}$ and generator system $\mathcal{G} = (R, P)$, we define

$$\mathrm{sat\_con}(\boldsymbol{v}, \mathcal{C}) \stackrel{\mathrm{def}}{=} \{\, c \in \mathcal{C} \mid \boldsymbol{v} \text{ saturates } c \,\};$$
$$\mathrm{sat\_gen}(c, \mathcal{G}) \stackrel{\mathrm{def}}{=} \{\, \boldsymbol{v} \in R \cup P \mid \boldsymbol{v} \text{ saturates } c \,\}.$$

Let $(\mathcal{C}, \mathcal{G}) \equiv \mathcal{R} \in \mathbb{CP}_{n+1}$ be such that $\mathcal{R} \Rrightarrow_\epsilon \mathcal{P}$. The set of *strict* and *non-strict inequality encodings* $\mathcal{C}_>$ and $\mathcal{C}_\geq$ of constraint system $\mathcal{C}$ are defined as

$$\mathcal{C}_> \stackrel{\mathrm{def}}{=} \left\{ \big(\langle \boldsymbol{a}, \boldsymbol{x}\rangle + s \cdot \epsilon \geq b\big) \in \mathcal{C} \;\middle|\; \boldsymbol{a} \neq \boldsymbol{0}, s < 0 \right\};$$
$$\mathcal{C}_\geq \stackrel{\mathrm{def}}{=} \left\{ \big(\langle \boldsymbol{a}, \boldsymbol{x}\rangle + s \cdot \epsilon \geq b\big) \in \mathcal{C} \;\middle|\; \boldsymbol{a} \neq \boldsymbol{0}, s = 0 \right\};$$

we also define the set of *ϵ-upper-bounds* as

$$\mathcal{C}_\epsilon \stackrel{\mathrm{def}}{=} \left\{ \big(\langle \boldsymbol{a}, \boldsymbol{x}\rangle + s \cdot \epsilon \geq b\big) \in \mathcal{C} \;\middle|\; \boldsymbol{a} = \boldsymbol{0}, s < 0 \right\}.$$

For ease of notation, a constraint $c \in \mathcal{C}_\epsilon$ will be usually denoted as $\epsilon \leq \delta$, where $\delta \stackrel{\mathrm{def}}{=} b/s$. Note that, by Proposition 3, we have $\delta > 0$ whenever $\mathcal{P} \neq \varnothing$. Similarly, the set of *closure point encodings* $\mathcal{G}_C$, the set of *point encodings* $\mathcal{G}_P$, and the set of *unmatched point encodings* $\mathcal{G}_U \subseteq \mathcal{G}_P$ of the generator system $\mathcal{G} = (R, P)$ are defined as follows:

$$\mathcal{G}_C \stackrel{\mathrm{def}}{=} \{\, (\boldsymbol{p}^{\mathrm{T}}, e)^{\mathrm{T}} \in P \mid e = 0 \,\};$$
$$\mathcal{G}_P \stackrel{\mathrm{def}}{=} \{\, (\boldsymbol{p}^{\mathrm{T}}, e)^{\mathrm{T}} \in P \mid e > 0 \,\};$$
$$\mathcal{G}_U \stackrel{\mathrm{def}}{=} \{\, (\boldsymbol{p}^{\mathrm{T}}, e)^{\mathrm{T}} \in P \mid e > 0, (\boldsymbol{p}^{\mathrm{T}}, 0)^{\mathrm{T}} \notin P \,\}.$$

Definition 6. (Strong minimal form.) *Let $\mathcal{R} \Rrightarrow_\epsilon \mathcal{P}$, where $(\mathcal{C}, \mathcal{G}) \equiv \mathcal{R}$ is a DD pair in minimal form. Then, we say that*

- *$\mathcal{C}$ is in* strong minimal form *if there does not exist a constraint system $\mathcal{C}'$ in minimal form such that $(\mathcal{C}'_> \cup \mathcal{C}'_\geq) \subset (\mathcal{C}_> \cup \mathcal{C}_\geq)$ and $\mathrm{con}(\mathcal{C}') \Rrightarrow_\epsilon \mathcal{P}$;*
- *$\mathcal{G} = (R, P)$ is in* strong minimal form *if there does not exist a generator system $\mathcal{G}' = (R', P')$ such that $\mathcal{G}' \neq \mathcal{G}$, $R' \subseteq R$, $P' \subseteq P$ and $\mathrm{gen}(\mathcal{G}') \Rrightarrow_\epsilon \mathcal{P}$.*

For the computation of strong minimal forms (smf's, for short), the key step is the identification of ϵ-redundant constraints and generators.

Definition 7. (ϵ-redundancy.) *Let $(\mathcal{C}, \mathcal{G}) \equiv \mathcal{R} \in \mathbb{CP}_{n+1}$. A constraint c is* ϵ-redundant *in $\mathcal{C}$ if $c \in \mathcal{C}_>$ and at least one of the following conditions holds:*

$$\mathrm{sat\_gen}(c, \mathcal{G}) \cap \mathcal{G}_C = \varnothing;$$
$$\exists c' \in \mathcal{C}_> \setminus \{c\} \,.\, \mathrm{sat\_gen}(c, \mathcal{G}) \setminus \mathcal{G}_P \subseteq \mathrm{sat\_gen}(c', \mathcal{G}).$$

A generator $\boldsymbol{p}$ is ϵ-redundant *in $\mathcal{G}$ if $\boldsymbol{p} \in \mathcal{G}_U$ and*

$$\exists \boldsymbol{p}' \in \mathcal{G}_P \setminus \{\boldsymbol{p}\} \,.\, \mathrm{sat\_con}(\boldsymbol{p}, \mathcal{C}) \cap \mathcal{C}_\geq \subseteq \mathrm{sat\_con}(\boldsymbol{p}', \mathcal{C}).$$

Proposition 4. *Let $\mathcal{R} \Rrightarrow_\epsilon \mathcal{P} \neq \varnothing$, where $(\mathcal{C}, \mathcal{G}) \equiv \mathcal{R}$ is a DD pair in minimal form. Then, the following hold:*

1. *If c is ϵ-redundant in $\mathcal{C}$, then* $\mathrm{con}\bigl(\mathcal{C} \setminus \{c\} \cup \{\epsilon \leq 1\}\bigr) \Rrightarrow_\epsilon \mathcal{P}$*;*
2. *If $\boldsymbol{p}$ is ϵ-redundant in $\mathcal{G} = (R, P)$, then* $\mathrm{gen}\Bigl(\bigl(R, P \setminus \{\boldsymbol{p}\}\bigr)\Bigr) \Rrightarrow_\epsilon \mathcal{P}$*;*
3. *If $\mathcal{C}$ contains no ϵ-redundant constraint, then it is in* smf*;*
4. *If $\mathcal{G}$ contains no ϵ-redundant generator, then it is in* smf*.*

It is worth stressing that, even if $(\mathcal{C}, \mathcal{G})$ is a DD pair for $\mathcal{R} \Rrightarrow_\epsilon \mathcal{P}$, after removing all ϵ-redundant constraints and generators the resulting descriptions $\mathcal{C}'$ and $\mathcal{G}'$ may be such that $\mathrm{con}(\mathcal{C}') \neq \mathrm{gen}(\mathcal{G}')$, so that $(\mathcal{C}', \mathcal{G}')$ is not a DD pair. Moreover, having one of the systems in smf does not imply that the corresponding dual description will also be in smf.

As an example, we now compute smf's for the polyhedron $\mathcal{R}_3$ represented in Figure 2(a). Let us first consider the constraint system. The two strict inequality encodings $x - \epsilon \geq 0$ and $-x - \epsilon \geq -3$, which correspond to segments $[O, B]$ and $[C, D]$, are not ϵ-redundant, because they are saturated by the closure point encodings O and C, respectively. In contrast, the constraint $x - 3\epsilon \geq 4$, corresponding to segment $[B, D]$, is identified as ϵ-redundant (no closure point encoding saturates it) and can be removed provided we add the constraint $\epsilon \leq 1$. The resulting constraint system, which is in smf, defines the trapezium of vertices O, C, E, and B represented in Figure 3(a). Note that the generator system for this trapezium is not in smf; this additional property can be enforced by removing the ϵ-upper-bound constraint $\epsilon \leq 1$, therefore obtaining the triangle of vertices O, C, and F.

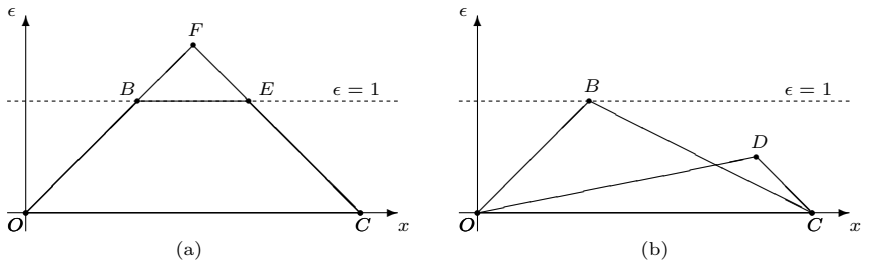


Fig. 3. Four different ϵ-representations for $\mathcal{P}_3$, obtained by applying the strong minimization process to (a) the constraint system; or (b) the generator system.

Starting again from polyhedron $\mathcal{R}_3$, let us now consider the strong minimization of its generator system, which is made up of the four points O, C, D, and B. Each one of the two unmatched point encodings B and D is made ϵ-redundant by the other one (they both saturate the empty set of non-strict inequality encodings); as a consequence, one of them can be removed, obtaining either one of the triangles OCB and OCD represented in Figure 3(b), which are both in smf.

5 The Parma Polyhedra Library

The Parma Polyhedra Library (PPL, `http://www.cs.unipr.it/ppl/`) is a collaborative project started in January 2001 at the Department of Mathematics of the University of Parma. It aims at becoming a truly professional library for the handling of approximations based on (not necessarily closed) convex polyhedra targeted at abstract interpretation and computer-aided verification. In this section we briefly review some of the key features of the library. Before continuing, it is perhaps worth stressing that the theoretical treatment of previous sections applies to polyhedra in $\mathbb{R}^n$. Not surprisingly, the implementation only deals with rational polyhedra, that is, polyhedra that can be defined by constraints with rational coefficients or, dually, generators with rational coordinates. It is easy to prove that the property of being rational is preserved by all the operations of interest for the applications the library aims at.

The library is written in standard C++ and this ensures, among other things, maximum portability across different computing platforms. Using C++ for the development made it easier to adopt a number of programming techniques that are the key to the library's robustness, generality, efficiency and usability. However, for the sake of maximal code reuse and utility, care has been taken not to require the client application to be written in C++. The library includes a complete C interface and thus can be interfaced to all programming languages' implementations (there are many) that provide a C interface. There is also a Prolog interface supporting several Prolog systems.

One of the key features of the library is robustness. In particular, this means that failure is avoided whenever possible and, in all other cases, failure is recognized and handled properly. The PPL uses arbitrary precision integer arithmetic to implement coefficients and coordinates, and is thus immune from both rounding and overflow problems. In addition, all the data structures used in the implementation are fully dynamic and can expand automatically (in amortized constant time) to any dimension allowed by the available virtual memory. Two other aspects of failure avoidance are hiding and systematic checking of the interface invariants. In contrast to other libraries, the PPL hides the implementation details almost completely. For instance, the internal representation of constraints, generators and systems thereof need not concern the client application. Similarly, implementation devices such as the *positivity constraint* [32] and all the matters regarding the ϵ-representation encoding of NNC polyhedra are completely invisible from outside. The client application is provided with more natural interfaces, allowing the direct manipulation of higher level concepts, such as inequalities, lines and closure points. For instance, in the appropriate contexts, '`X < 5*Z`' and '`X + 2*Y + 5*Z >= 7`' is valid syntax expressing a strict and a non-strict inequality, both in the C++ and the Prolog interfaces. Even the C interface, which is at a considerably lower level of abstraction, refers to concepts like linear expression, constraint and constraint system and not to their possible implementations such as vectors and matrices. As usual, this is important for error prevention and to allow maximum latitude for the implementation.

Forbidding access to the internal structures manipulated by the library and systematic checking of the interface invariants could impede the overall system

performance. However, the resulting overhead can be completely repaid (to the point of giving rise to a speedup) if the implementation exploits the freedom it has from the user interface.

In the design of the library, particular care has been taken concerning issues of scalability since any user of any polyhedra library must face the possibility of excessive CPU time consumption and/or memory usage [23]. A far from satisfactory but possible solution is to hope for the best but eventually "kill" those processes requiring more than the available resources [7]. In contrast, what we aim at is support for the dynamic and automatic composition of the trade-off between expressivity and efficiency. In this scenario, the static analysis or verification procedures should, by default, use the more descriptive (and costly) domains, such as the convex polyhedra. When the computation of an operation on these descriptions requires too much time, or memory space, the system should detect this and perform a change of representation, for instance by approximating the input polyhedra with enclosing *bounding boxes* (rectangular regions with sides parallel to the axes); after the execution of the requested operation on this less precise (but much less computationally expensive) domain, the result is converted back into a convex polyhedron. Such a scenario is feasible only if the polyhedra library is implemented so that it can consistently react, without any loss of system resources, to events such as a timeout or an exception thrown by the memory allocation routines. As far as we know, the Parma Polyhedra Library is the first one to provide this facility. The idea is exemplified in Figure 4, where a robust C++ implementation of `complex_function` is sketched. The objective is to compute the polyhedron `result` starting from the polyhedra

```
typedef Parma_Polyhedra_Library::NNC_Polyhedron PH;
void complex_function(const PH& ph1, const PH& ph2, PH& result) {
  try {
    start_timer(max_time_for_complex_function);
    complex_function_on_polyhedra(ph1, ph2, result ...);
    stop_timer();
  }
  catch (Exception& e) { // Memory exhausted or timeout or other error.
    ...
    BoundingBox bb1, bb2, bb_result;
    ph1.shrink_bounding_box(bb1);  ph2.shrink_bounding_box(bb2);
    complex_function_on_bounding_boxes(bb1, bb2, bb_result ...);
    result = Polyhedron(bb_result);
  }
}
```

Fig. 4. Falling back to bounding boxes when the analysis with polyhedra *is* too costly.

`ph1` and `ph2`. The operation is first tried on the polyhedra themselves and, in case of failure, the computation is done on the bounding box approximations of `ph1` and `ph2`. In the latter case the result of the bounding box computation,

`bb_result` is converted into a convex polyhedron object and assigned to the output polyhedron `result`. The trade-off between efficiency and precision can also be controlled by means of the widening operations provided by the PPL, which are based on those originally proposed in [15] and improved in [14].

Concerning the efficiency of the PPL, at present no application can use both the PPL and another convex polyhedra library, so comparisons are not possible. However, the PPL has been integrated with the China analyzer [1] for the purpose of detecting linear argument size relations [5]. The performance of the combined system has been compared, on the same task, with the performance of the cTI analyzer [28], which uses an implementation of convex polyhedra based on the SICStus CLP(Q) package. The combined system China+PPL outperformed that version of cTI in a significant way, exhibiting termination instead of thrashing and speedups of one or more orders of magnitude.

The Parma Polyhedra Library is free software released under the GPL: code and documentation can be downloaded and its development can be followed at `http://www.cs.unipr.it/ppl/`.

6 Conclusion

Convex polyhedra provide the basis for several abstractions used in static analysis and computer-aided verification of complex and sometimes mission critical systems. For such purposes an implementation of convex polyhedra must be firmly based on a clear theoretical framework and written in accordance with sound software engineering principles. In this paper we have presented some of the most important ideas that are behind the Parma Polyhedra Library. In particular, we have provided a novel theoretical framework for the representation and manipulation of not necessarily closed convex polyhedra. We have also briefly sketched some important features of the design and implementation of the library. We believe that all this work constitutes a significant improvement in the state of the art and an encouragement to the wider adoption of abstract interpretation techniques in static analysis and computer-aided verification.

Acknowledgments. We would like to thank Costantino Medori for the many useful discussions we had with him on the subject of this paper, Fred Mesnard for letting us play with cTI's analyzer, and the anonymous reviewers for their helpful comments and observations.

References

1. R. Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages.* PhD thesis, Dipartimento di Informatica, Università di Pisa, Pisa, Italy, 1997. Printed as Report TD-1/97.
2. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. *The Parma Polyhedra Library User's Manual.* Department of Mathematics, University of Parma, Parma, Italy, release 0.4 edition, July 2002. Available at `http://www.cs.unipr.it/ppl/`.

3. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. Quaderno 286, Dipartimento di Matematica, Università di Parma, Italy, 2002. See also [4]. Available at `http://www.cs.unipr.it/Publications/`.
4. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Errata for technical report "Quaderno 286". Available at `http://www.cs.unipr.it/Publications/`, 2002.
5. F. Benoy and A. King. Inferring argument size relationships with CLP(R). In J. P. Gallagher, editor, *Logic Programming Synthesis and Transformation: Proceedings of the 6th International Workshop*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–223, Stockholm, Sweden, 1997. Springer-Verlag, Berlin.
6. F. Besson, T. P. Jensen, and J.-P. Talpin. Polyhedral analysis for synchronous languages. In A. Cortesi and G. Filé, editors, *Static Analysis: Proceedings of the 6th International Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 51–68, Venice, Italy, 1999. Springer-Verlag, Berlin.
7. N. S. Bjørner, A. Browne, M. Colón, B. Finkbeiner, Z. Manna, M. Pichora, H. B. Sipma, and T. E. Uribe. *STeP: The Stanford Temporal Prover (Educational Release) User's Manual*. Computer Science Department, Stanford University, Stanford, California, version 1.4-α edition, July 1998. Available at `http://www-step.stanford.edu/`.
8. N. V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of system of linear equations. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 4(4):151–158, 1964.
9. N. V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of system of linear inequalities. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 5(2):228–233, 1965.
10. N. V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.
11. M. A. Colón and H. B. Sipma. Synthesis of linear ranking functions. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *Lecture Notes in Computer Science*, pages 67–81, Genova, Italy, 2001. Springer-Verlag, Berlin.
12. P. Cousot, editor. *Static Analysis: 8th International Symposium, SAS 2001*, volume 2126 of *Lecture Notes in Computer Science*, Paris, 2001. Springer-Verlag, Berlin.
13. 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 the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
14. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295, Leuven, Belgium, 1992. Springer-Verlag, Berlin.
15. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, Tucson, Arizona, 1978. ACM Press.
16. N. Dor, M. Rodeh, and S. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In Cousot [12], pages 194–212.

17. K. Fukuda and A. Prodon. Double description method revisited. In M. Deza, R. Euler, and Y. Manoussakis, editors, *Combinatorics and Computer Science, 8th Franco-Japanese and 4th Franco-Chinese Conference, Brest, France, July 3-5, 1995, Selected Papers*, volume 1120 of *Lecture Notes in Computer Science*, pages 91–111. Springer-Verlag, Berlin, 1996.
18. N. Halbwachs. Delay analysis in synchronous programs. In C. Courcoubetis, editor, *Computer Aided Verification: Proceedings of the 5th International Conference*, volume 697 of *Lecture Notes in Computer Science*, pages 333–346, Elounda, Greece, 1993. Springer-Verlag, Berlin.
19. N. Halbwachs, A. Kerbrat, and Y.-E. Proy. *POLyhedra INtegrated Environment*. Verimag, France, version 1.0 of POLINE edition, September 1995. Documentation taken from source code.
20. N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In B. Le Charlier, editor, *Static Analysis: Proceedings of the 1st International Symposium*, volume 864 of *Lecture Notes in Computer Science*, pages 223–237, Namur, Belgium, 1994. Springer-Verlag, Berlin.
21. N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.
22. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1(1+2):110–122, 1997.
23. T. A. Henzinger, J. Preussig, and H. Wong-Toi. Some lessons from the HYTECH experience. In *Proceedings of the 40th Annual Conference on Decision and Control*, pages 2887–2892. IEEE Computer Society Press, 2001.
24. B. Jeannet. *Convex Polyhedra Library*, release 1.1.3c edition, March 2002. Documentation of the "New Polka" library available at `http://www.irisa.fr/prive/Bertrand.Jeannet/newpolka.html`.
25. H. Le Verge. A note on Chernikova's algorithm. *Publication interne* 635, IRISA, Campus de Beaulieu, Rennes, France, 1992.
26. V. Loechner. *PolyLib*: A library for manipulating parameterized polyhedra. Available at `http://icps.u-strasbg.fr/~loechner/polylib/`, March 1999. Declares itself to be a continuation of [32].
27. Z. Manna, N. S. Bjørner, A. Browne, M. Colón, B. Finkbeiner, M. Pichora, H. B. Sipma, and T. E. Uribe. An update on STeP: Deductive-algorithmic verification of reactive systems. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computing Sciences. Springer-Verlag, Berlin, 1999.
28. F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. In Cousot [12], pages 93–110.
29. T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall. The double description method. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games – Volume II*, number 28 in Annals of Mathematics Studies, pages 51–73. Princeton University Press, Princeton, New Jersey, 1953.
30. W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.
31. J. Stoer and C. Witzgall. *Convexity and Optimization in Finite Dimensions I*. Springer-Verlag, Berlin, 1970.
32. D. K. Wilde. A library for doing polyhedral operations. Master's thesis, Oregon State University, Corvallis, Oregon, December 1993. Also published as IRISA *Publication interne* 785, Rennes, France, 1993.

Speeding Up Dataflow Analysis Using Flow-Insensitive Pointer Analysis

Stephen Adams[1], Thomas Ball[1], Manuvir Das[1], Sorin Lerner[2], Sriram K. Rajamani[1], Mark Seigle[2], and Westley Weimer[3]

[1] Microsoft Research
[2] University of Washington
[3] UC Berkeley

Abstract. In recent years, static analysis has increasingly been applied to the problem of program verification. Systems for program verification typically use precise and expensive interprocedural dataflow algorithms that are difficult to scale to large programs. An attractive way to scale these analyses is to use a preprocessing step to reduce the number of dataflow facts propagated by the analysis and/or the number of statements to be processed, before the dataflow analysis is run. This paper describes an approach that achieves this effect. We first run a scalable, control-flow-insensitive pointer analysis to produce a conservative representation of value flow in the program. We query the value flow representation at the program points where a dataflow solution is required, in order to obtain a conservative over-approximation of the dataflow facts and the statements that must be processed by the analysis. We then run the dataflow analysis on this "slice" of the program.
We present experimental evidence in support of our approach by considering two client dataflow analyses for program verification: typestate analysis, and software model checking. We show that in both cases, our approach leads to dramatic speedups.

1 Introduction

In recent years, static analysis has increasingly been applied to the problem of program verification. Two kinds of algorithms for static program analysis have been proposed. Flow-insensitive algorithms such as [DLFR01,HT01,OJ97] scale to large programs, but do not handle strong updates precisely. Flow-sensitive algorithms [CRL99,WL95], on the other hand, can handle strong update precisely but do not scale to large programs. The absence of strong update adversely affects software engineering and program verification tools, which produce many false positives with conservative analysis. Therefore, these tools often employ costly interprocedural dataflow algorithms, even though this choice precludes application to large programs. This paper is based on two key insights. The first insight is that the points-to algorithm of Das [Das00] can be viewed as producing a lightweight, conservative representation of the value flow in a program, called the value-flow graph (VFG). The second insight is that by querying the

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 230–246, 2002.
© Springer-Verlag Berlin Heidelberg 2002

VFG before a client dataflow analysis is run, both the number of dataflow facts propagated by the analysis and the part of the program that must be analyzed can be reduced.

Two small examples that demonstrate our approach are given in Figure 1.

Figure 1(a) shows a procedure whose file I/O related behavior must be summarized for all callers. By performing a "typestate slice" in the VFG, we can remove dataflow facts corresponding to the states of the file handle `g2` (since the state of `g2` is not changed by `bar`) and statements that do not affect the state of `g1`, before analyzing `bar`.

Figure 1(b) shows a program fragment on which we wish to determine whether the `abort` function is called. One way to do this is to determine a set of predicates over program variables, construct a boolean program abstraction [BR01a] over these predicates, and then model check the abstraction using dataflow analysis. The predicates required for proving that the `abort` statement is unreachable are `x == 5`, `y == 5`, `x == 10`, and `y == 10`. These predicates can be discovered in two iterations of iterative refinement (see Section 4). By performing a "predicate slice" in the VFG, we can infer the set of constants that flow into variables `x` and `y` and generate these predicates in advance, eliminating the iteration overhead.

These examples show the sort of information we hope to provide to client dataflow analyses. The typestate slice and predicate slice can both be obtained from the VFG. Each slice reduces the workload of the client dataflow analysis.

Two alternative approaches for achieving the same effect are demand-driven versions of the client dataflow analyses or standard program slicing. Both approaches have drawbacks. Frameworks for demand-driven dataflow [DGS95, HRS95] do not handle a general enough class of dataflow problems to allow application to problems that involve value flow. Program slicing [HRB90], which is based on dataflow analysis, it typically too expensive to apply on large programs.

```
(a)  FILE *g1, *g2;
     void bar() {
       FILE *l, *m;
       l = g1;
       m = g2;
       fclose(l);
     }
```

```
(b)  void baz(bool b) {
       int x , y;
       if (b) {
         x = 5;
         y = 5;
       } else {
         x = 10;
         y = 10;
       }
       if (x != y)
         abort();
     }
```

Fig. 1. Examples of speeding up dataflow analysis. Figure (a) above shows a fragment of C code to be sliced w.r.t. file I/O behavior. Figure (b) above shows a fragment of C code to be sliced w.r.t. reachability of calls to `abort`.

The main contributions of this paper are:

- We show how the flow-insensitive pointer analysis of Das can be used to produce a program-point-independent graph representation of value flow (the VFG) in a program.
- We show how the VFG can be used to perform client-specific program slices that significantly speed up client dataflow analyses.
- We demonstrate the effectiveness of our technique with two specific applications of slicing using the VFG:
  - Typestate slicing: We use typestate slicing to reduce the number of objects whose state is tracked by ESP [DLS02], a typestate checker for large programs. We apply ESP to check file I/O properties of the `gcc` compiler. Typestate slicing reduces the average number of objects tracked per procedure from 1100 to less than 1, making the dataflow analysis in ESP practical.
  - Predicate slicing: We use predicate slicing to generate a candidate set of predicates for the initial boolean program abstraction used by SLAM [BR01a], a software model checker. We apply SLAM to check properties of several Windows device drivers. Predicate slicing reduces the running time of SLAM by a factor of 2-10, and allows SLAM to terminate in some cases where it did not terminate before.

The rest of this paper is structured as follows. In Section 2, we describe our value flow representation, its computation, and its interface to dataflow clients. We then present two concrete applications of our value flow slicing approach. In Section 3, we demonstrate the use of typestate slicing in ESP. In Section 4, we show how predicate slicing can be used to aid predicate discovery in SLAM. We discuss related work in Section 5, and conclude in Section 6.

2 Value Flow via Pointer Analysis

Pointer analysis algorithms typically produce graph representations ("points-to graphs") of pointer relationships in programs. These graphs encode information about which memory locations hold references to other memory locations. More importantly, algorithms based on subtyping encode constraints that arise from value flow through assignments in the program: every assignment that causes flow of pointer values is represented either implicitly or explicitly in the graph. The key insight of this paper is that if pointer analyses based on subtyping are modified to process all assignments, rather than just pointer assignments, and if constant values are represented explicitly in the graph, the resulting points-to graph encodes a conservative approximation of all value flow in the program (we call this the value flow graph, or VFG). In addition, if the fragments of the graph that result from processing constraints are labeled with the identities of the statements that generated the constraints, the VFG encodes slices of the program: each slice represents the set of program statements that contribute to the value of a given expression. In the terminology of program slicing [Tip95],

these slices encode flow dependences, but not control dependences. When the client dataflow analysis ignores control dependences (this is typically true of many dataflow analyses, including the examples considered in this paper) the flow dependence slices encoded in the VFG preserve the results produced by the client dataflow analysis.

We focus on flow-insensitive analyses, which produce a single representation of all of the value flow in the program that is applicable at every point in the program. Flow-sensitive analyses could be used to produce VFGs as well: they would produce separate graphs at each program point, though every graph needs to encode only values relevant at the program point.

2.1 One-Level Value-Flow Analysis

In this paper, we use the one-level-flow subtyping analysis of Das [Das00] to produce value flow information. The points-to graphs produces by this algorithm make value flow through assignments explicit, via so-called "flow" edges.

The graph includes two kinds of nodes: source nodes, and expression nodes. Source nodes represent variables in the program, and are labeled by the variable name. Expression nodes represent pointer dereferences of source nodes. An expression node can be named by a path starting from a source node with a dereference (*) prefixed for every points-to edge on the path from the source node. Due to aliasing, expression nodes do not have unique names, and are thus not labeled.

The graph includes two kinds of edges: points-to edges (written $\rightarrow$) and flow edges (written $\rightarrow_F$). A points-to edge connects (the node of) an expression `e` with (the node of) an expression `*e`. A flow edge $*$ `e1` $\rightarrow_F$ $*$`e2` encodes an assignment from `e1` to `e2` in the program. Because the algorithm sometimes merges

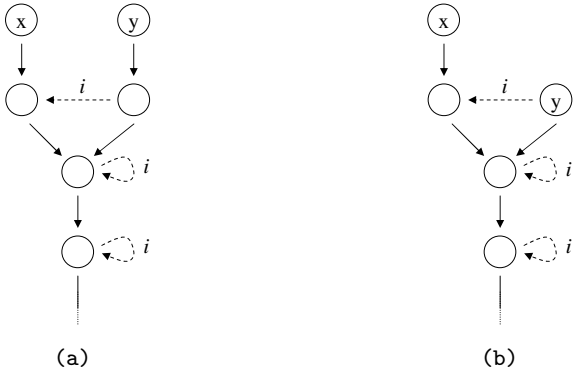


Fig. 2. Figures (a) and (b) above show the VFGs computed by the one level flow algorithm for i: `x = y` and i: `x = &y`, respectively. Points-to edges are solid arrows. Flow edges are labeled dashed arrows.

expressions into the same node, value flow is also encoded in node equality: every node can be viewed as having a flow edge to itself.

The graph fragments produced by the analysis for two kinds of assignments are shown in Figure 2. The assignment `x = y` induces an edge $*\mathtt{y} \rightarrow_F *\mathtt{x}$, indicating that the value of `y` may flow to `x`, and induces node equality for `**x` and `**y`, `***x` and `***y`, and so on, indicating that the assignment also implicitly creates value flow between `*x` and `*y`, `**x` and `**y`, and so on. The assignment `x = &y` induces an edge $\mathtt{y} \rightarrow_F *\mathtt{x}$, indicating that the address of `y` may flow to `x`. A more detailed discussion of the algorithm can be found in [Das00].

We extend the analysis in two ways to create the VFG: First, we add nodes for values that are not stored in any memory location (*e.g.* constants). Second, we label flow edges with the assignments that induced the edges. We also transitively label all of the self-loop flow edges that are induced at lower (in terms of points-to edges) levels of the graph.

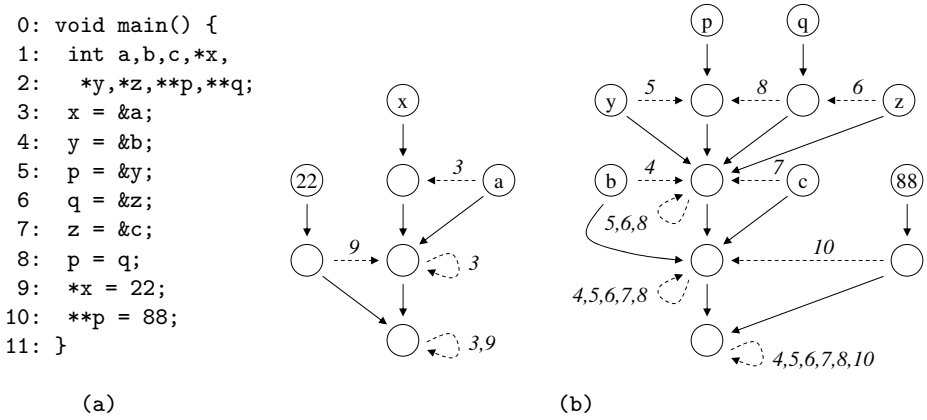


Fig. 3. (a) A fragment of C code and its VFG, augmented with source code labels and nodes for constants, and (b) its value-flow graph.

Example 1 A small C program is shown in Figure 3(a). The VFG computed for this program by the one-level value-flow analysis is shown in Figure 3(b). This graph encodes all of the value flow in the program. For instance, the flow of the value 22 to variable `a` at line 9 is encoded in the flow edge from `*22` to `*a` labeled 9. Notice that the statement at line 3, where `x` is assigned the address of `a`, also contributes to this value flow. The role of this statement is captured by the self-loop flow edge on `*a` labeled 3. □

For our purposes, the VFG produced by the modified one-level-flow algorithm has several interesting properties:

– The graph can be computed in almost-linear time.

- Points-to sets can be obtained from the graph in linear time: the set of variables pointed-to by p (written $\mathsf{Vars}(*\mathtt{p})$) may be obtained by performing backwards reachability along flow edges from *p and including all of the variables encountered along the way.
- Value flow can be queried in linear time: the set of expressions whose values may flow to e may be obtained by performing backwards reachability along flow edges from *e and including all of the expressions with points-to edges to nodes encountered along the way.

Example 2 The VFG in Figure 3 can be used to answer a number of points-to and value flow queries. A backwards flow query from **x picks up *22, which means that the value 22 can flow to any variable in the points-to set *x, including a. Because the query does not pick up any of b, c, y, z, p, or q, all of these variables can be ignored by a dataflow client that is tracking the flow of the value 22 through the program. □

2.2 Slicing Interface

Although the entire value flow graph could be presented to a client analysis, in practice most clients make a series of structured queries to the graph. This interface to the value flow graph is summarized below:

- $\mathsf{N}(\mathtt{v})$ is the node representing v, $\forall \mathtt{v} \in \mathit{ProgVar}$.
- $\mathsf{Deref}(\mathtt{p}) = \mathtt{q}$ where $\mathtt{p} \rightarrow \mathtt{q}$.

  $\mathsf{Deref}(\mathtt{p})$ performs a single pointer dereference from p in the VFG.
- $\mathsf{N}(\mathtt{e}) = \begin{cases} \mathsf{N}(\mathtt{v}) & \text{if } \mathtt{e} = \mathtt{v} \\ \mathsf{Deref}(\mathsf{N}(\mathtt{e}')) & \text{if } \mathtt{e} = *\mathtt{e}' \end{cases}$

  $\mathsf{N}(\mathtt{e})$ maps expression e to its node in the VFG.
- $\mathsf{Vars}(\mathtt{e}) = \{\mathtt{v} \in \mathit{ProgVar} \mid \mathtt{v} \rightarrow_F^* \mathtt{e}\}$,

  $\mathsf{Vars}(\mathtt{e})$ is the set of variables that may be aliases of expression e.
- $\mathsf{FlowsInto}(\mathtt{e}) = \{\mathtt{q} \mid \mathsf{Deref}(\mathtt{q}) \rightarrow_F^* \mathsf{Deref}(\mathtt{e})\}$.

  $\mathsf{FlowsInto}(\mathtt{e})$ returns all expressions whose values may flow into e.
- $\mathsf{FlowsTo}(\mathtt{e}) = \{\mathtt{q} \mid \mathsf{Deref}(\mathtt{e}) \rightarrow_F^* \mathsf{Deref}(\mathtt{q})\}$.

  Conversely, $\mathsf{FlowsTo}(\mathtt{e})$ returns all expressions into which the value of e may flow.

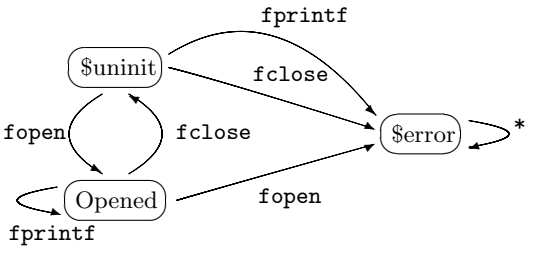
The case studies presented in the next two sections use this interface to the value flow graph. In this description we have not considered objects with structure. The value flow representation can be extended in a straightforward manner to encode structure field accesses.

3 Typestate Slicing in ESP

ESP [DLS02] is a verification tool that identifies violations of programmer specified properties in large C programs. ESP tracks "typestate" [SY86]: As a program executes, it creates values. Every value is associated with a type that is invariant during program execution. Some values are additionally associated with a state that may be updated by certain operations on values. Transitions between states are encoded in a state machine. The machine has a special error state; transitions to the error state indicate violations of the property. An example of a user specified property (valid file output) is given in Figure 4(a).

The core engine of ESP is an interprocedural dataflow analysis that computes the possible states of every value at every point in the program. Because ESP is intended for application on large programs, the dataflow analysis is performed bottom-up on the call graph, one strongly connected component at a time. As a result, ESP must summarize the typestate behavior of a function `foo` for all possible callers before any of the callers of `foo` are analyzed. The typestate summary of a function `foo` includes (a) the state changes caused by execution of `foo` on values live before the execution of `foo`, and (b) any new stateful values created by execution of `foo`, and their states on exit from `foo`. In order to compute the summary of a function, ESP first computes the set of memory locations that may hold stateful values at entry to the function, and then performs dataflow analysis for each such value. In a language with type coercion such as C, if ESP is computing the states of values of a particular type, for instance file handles with type `FILE *`, it must also consider values of all other possible types.

The set of locations that may hold a stateful value at entry to a function `foo` includes all globals, formal parameters of `foo`, and transitive dereference targets of these. We refer to this set as *inNodes* (input interface nodes):



(a)

```
FILE *e, *f, *g, *h;

void foo () {
   int *x, y;
   y = 3;  x = &y;
   g = fopen(...);
   bar(x);
}

void bar (int* p) {
   fprintf (h, "%d", *p);
   fclose (h);
}
```

(b)

Fig. 4. (a) A finite state property that specifies correct usage of a file output library, and (b) an example program that uses the library. Function `foo` calls `bar` with an integer pointer. `bar` prints its dereferenced parameter to `h` and then closes `h`.

$$inNodes(f) = \bigcup_{p \in params(f)} reached(p) \cup \bigcup_{g \in globals} reached(g)$$
$$reached(v) = \{v\} \cup reached(\mathsf{Deref}(v))$$
$$reached_p(v) = reached(v) - \{v\}$$

Similarly, the set of locations that may hold a freshly created value at exit from `foo` includes all globals, the return value of `bar`, and dereference targets of these and the formal parameters of `bar`. We refer to this set as *outNodes* (output interface nodes):

$$outNodes(f) = reached(return_f) \cup \bigcup_{p \in params(f)} reached_p(p) \cup \bigcup_{g \in globals} reached(g)$$

Example 3 A small program that manipulates file handles is given in Figure 4(b). The sets of possible interface nodes for the functions in this program are:

inNodes(`foo`): {`e, *e, f, *f, g, *g, h, *h`}
outNodes(`foo`): {`e, *e, f, *f, g, *g, h, *h`}
inNodes(`bar`): {`e, *e, f, *f, g, *g, h, *h, p, *p`}
outNodes(`bar`): {`e, *e, f, *f, g, *g, h, *h, *p`}

In large programs with many globals, the interface node sets can be large enough to make typestate checking infeasible. Clearly, these sets include many locations that contain values whose state is not changed by the called function. A preprocessing step that can eliminate many of the spurious locations from the interface node sets will increase the efficiency of the subsequent typestate analysis.

3.1 Eliminating Interface Nodes via Value Flow

ESP uses a slicing procedure over the VFG to eliminate nodes from *inNodes* before the typestate analysis is run. The slicing procedure is based on the following observation: A location `l` must be included in *inNodes(*`f`*)* only if there is some expression `e` such that `e` is an argument to a state changing operation performed during execution of `f`, and the value held by `l` at entry to `f` can flow to `e`. Therefore, we can query the VFG to obtain an over-approximation of the set of locations that must be included in *inNodes(*`f`*)*. We refer to this procedure as "typestate slicing".

A similar procedure can be used to eliminate nodes from *outNodes*. A location `l` must be included in *outNodes(*`f`*)* only if there is some expression `e` such that `e` is the result of a value creation operation, and the value of `e` can be held by `l` at exit from `f`.

ESP uses a language of syntactic patterns to identify state changing operations in the code. The control flow graphs produced by the ESP front-end contain two kinds of distinguished, mutually exclusive, pattern nodes:

- PATTERN(name, p): Represents a call to a function whose name appears along a transition in the protocol, i.e. `fclose(f)`. `p` is the expression on which the operation is applied.

– CPATTERN(name, p): Represents a call to a function which creates fresh stateful values, i.e. `f = fopen(...)`. In this case `p` represents the recipient of the new value.

The set of expressions that are arguments to pattern nodes can then be defined as follows:

$$pNodes(f) = \{n | PATTERN(\_, n) \in f\}$$
$$cpNodes(f) = \{n | CPATTERN(\_, n) \in f\}$$

The sliced sets $inNodes_s$ and $outNodes_s$ are given by:

$$inTargets(f) = pNodes(f) \cup \bigcup_{g \in callees(f)} inNodes_s(g)$$
$$inNodes_s(f) = inNodes(f) \cap \bigcup_{n \in inTargets(f)} \mathsf{FlowsInto}(n)$$

$$outSrcs(f) = cpNodes(f) \cup \bigcup_{g \in callees(f)} outNodes_s(g)$$
$$outNodes_s(f) = outNodes(f) \cap \bigcup_{n \in outSrcs(f)} \mathsf{FlowsTo}(n)$$

The equations above describe a slicing procedure that is applied bottom-up on the call graph, one strongly connected component (SCC) at a time. Although the equations appear to require a fixpoint computation for SCCs containing more than one function, the solution can be obtained by combining a single flow query each for $inNodes_s$ and $outNodes_s$ with intersection operations for each function in the SCC.

Example 4 After typestate slicing is applied, the sets of interface nodes for the functions in the program from Figure 4(b) are:

$inNodes$(`foo`): {`h`} $\quad$ $outNode$(`foo`): {`g`}
$inNodes$(`bar`): {`h`} $\quad$ $outNodes$(`bar`): {}

3.2 Experiments

We have implemented typestate slicing as a preprocessing step in ESP. The dataflow engine in ESP performs an exhaustive dataflow analysis of the entire code in a strongly connected component for every node in the interface node sets of functions in the SCC. Therefore, the performance of ESP is directly related to the number of interface nodes. ESP also uses typestate slicing to dramatically reduce the number of relevant CFG nodes before dataflow analysis. We do not discuss those results here.

File output in gcc. ESP has been applied to the problem of verifying the file output behavior of a version of the `gcc` compiler, taken from the SpecInt95 benchmark suite, using the property specification given in Figure 4. `gcc` has 140,000 LOC in 2149 functions over 66 files; there are 1,086 global and static variables; the call graph contains a single SCC with over 450 functions.

Typestate slicing applied bottom-up on `gcc` requires 200 seconds on a Toshiba Tecra 8200 laptop with a 1GHz Pentium III processor and 512MB RAM, running Windows XP. Slicing reduces the number of interface nodes from roughly 1100 to < 1 on average, with a median of 15 for functions with non-empty interface node sets. This dramatic reduction in the sizes of the interface node sets allows ESP to successfully verify `gcc` in less than 15mins and 750MB of memory. Verification of `gcc` w.r.t. the file output property would be infeasible if the set of global interface nodes were not pruned.

Registry key leakage in cmd. We have also used ESP to find resource leaks in the command shell interpreter (`cmd`) of a version of the Windows operating system. `cmd` has 40,000 LOC; there are 483 global and static variables. Typestate slicing applied bottom-up on `cmd` requires 33 seconds on a Compaq Evo W6000 desktop PC with a 2.2GHz Pentium IV processor and 2GB RAM, running Windows XP. Slicing reduces the number of interface nodes from roughly 500 to $<< 1$ on average, with a median of 1 for functions with non-empty interface node sets.

4 Predicate Slicing in SLAM

4.1 SLAM Overview

The SLAM toolkit validates temporal safety properties of C programs through a process of boolean abstraction [BMMR01,BR01a], interprocedural dataflow analysis, and counterexample-driven refinement. The first step in this process involves abstracting the C program to a boolean program. Boolean programs have all of the control structure of C programs but contain only boolean variables. These boolean variables represent predicates over expressions in the original program. For example, a boolean variable might represent "`(*ptr)==2`". The soundness of the boolean abstraction means that if a variable "`(*ptr)==2`" is true at a point L in the boolean program then `(*ptr)==2` will be true at the same point L in the original C program. Given a set of predicates $\mathcal{P}$ and a C program, SLAM generates the corresponding boolean program. A key property of this transformation is that if the error state is *not* reachable in the boolean program then it is *not* reachable in the original program. Assuming $\mathcal{P}$ has been well-chosen, checking the safety of the original program reduces to checking the safety of the boolean program.

Since the boolean program involves only control flow (`if`, `goto`, function calls) and a finite set of boolean variables, interprocedural dataflow analysis can be used to explore its state-space exhaustively [BR01b].[1] If the error state is not reachable, the program adheres to the safety policy. If there is a path to the error state then the path can be checked for feasibility in the original C program. If the path is feasible then the SLAM produces an error trace demonstrating how the original program violates the safety policy. Otherwise, the path is an

[1] When viewed as a dataflow problem, $\mathcal{P}$ is the set of dataflow facts and the boolean abstraction process gives the transition function for every statement in the program.

infeasible counterexample (or false positive) and SLAM generates new predicates to increase the precision of the boolean program abstraction on the subsequent iterations of SLAM.

4.2 Better Predicate Generation

The SLAM process is an example of "counterexample-driven refinement". When SLAM was first implemented, the initial set of predicates $\mathcal{P}$ was the empty set, and counterexample-driven refinement was used to expand the set of predicates as necessary. This process required many iterations in some cases (over 20 iterations for some Windows device drivers).

If, instead, we could start SLAM off with a "good" set of predicates that included most (if not all of the predicates) that would be discovered by the counterexample-driven refinement process, then SLAM would terminate more quickly. On the other hand, if we start with too many predicates (for example, all boolean relations between all pairs of expressions in the program) the search space may be too large and SLAM's dataflow analysis might exhaust system resources. Such a general approximation of predicates would link variables that are never meaningfully related by the flow of values in the original program.

We present a predicate slicing algorithm that works for a restricted subset of the C language. Given a VFG G for a program written in that subset, the algorithm produces a set of predicates $\mathcal{P}$ that is provably sufficient to avoid false positives but avoids linking unrelated terms. For C programs that fall outside this restricted subset, the algorithm produces a good set of initial predicates, and the remaining false positives can be eliminated using iterative refinement. This algorithm has been used in practice to produce predicate sets that allow for the rapid analysis of programs for which the naive analysis was either very slow or infeasible. By using the VFG, we were able to hit a "sweet-spot" and generate an initial set of predicates detailed enough to eliminate false positives but small enough to make the analysis scale.

4.3 Input Language

For the purposes of presentation, we consider a restricted subset of C which contains local scoping, procedural abstraction and the following statements:

$$
\begin{array}{rll}
s ::= & v_i \leftarrow n & (n \in \mathcal{Z}) \\
| & v_i \leftarrow v_j & \\
| & \mathsf{if}\ (\star)\ s_1\ \mathsf{else}\ s_2 & \\
| & v_i \leftarrow \mathsf{fun}(v_j, ...) & \\
| & \mathsf{return}(v_j) & \\
| & \mathsf{abortif}(v_i \approx v_j) &
\end{array}
$$

v_i and v_j represent variables. All non-parameter variables are assumed to be initialized before use. Integer constants n form the set of *ground terms* in the program. The $\mathsf{if}\ (\star)$ construct represents a non-deterministic `if`. The $\mathsf{abortif}(v_i \approx v_j)$ statement represents the safety policy: if such a statement can be reached

when $(v_i \approx v_j)$ is true, the program violates the safety property. The relational operator $\approx$ is left abstract but is assumed to be deterministic. While severely restricted, this language is motivated by device driver handling of status codes. The codes are enumerated constants defined in header files and no transforming operations (e.g., plus, bitwise-and) may legally be performed on them. However, the safety policy may insist that status codes be propagated or that certain functions be called (or not called) depending on status code values. In such a device driver example the set of ground terms would be the set of status code values considered by the program.

4.4 Predicate Slicing Algorithm

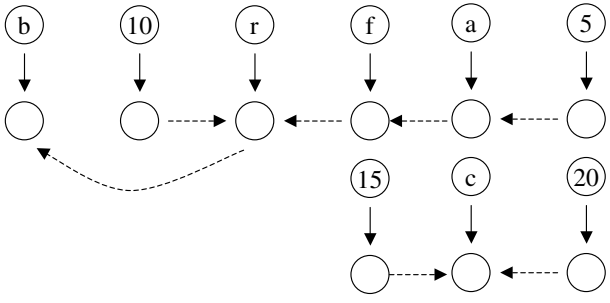
Given a program C in this language, let G be its VFG. Intuitively, the algorithm uses G to track all variables and ground terms that can flow into v_i and v_j for every $\mathsf{abortif}(v_i \approx v_j)$ statement and generates predicates to keep track of that flow of values.

Ideally the algorithm would emit the predicate "a==b" if $\mathsf{Deref}(\mathsf{N}(\mathtt{a})) \rightarrow_F \mathsf{Deref}(\mathsf{N}(\mathtt{b})) \in G$. Unfortunately, such predicates are often meaningless. For example, in Figure 5, although $\mathsf{Deref}(\mathsf{N}(\mathtt{r})) \rightarrow_F^* \mathsf{Deref}(\mathsf{N}(\mathtt{b}))$ will be in G, the predicate "r==b" cannot be interpreted at any point in the program since r is a local variable in select and b is a local variable in main. That predicate is thus not suitable as a dataflow fact for the SLAM toolkit. A similar scoping problem occurs between actual arguments and formal parameters.

We will use the set of ground terms (constant values) to bridge the gap between variables in different scopes. For example, given that the predicates

```
int select(int f)
{
  int r;
  if (*) r ← f;
  else r ← 10;
  return(r);
}
void main() {
  int a,b,c;
  a ← 5;
  b ← select(a);
  if (*) c ← 15;
  else c ← 20;
  abortif(b > c);
}
```

(a)



(b)

Fig. 5. An example of predicate slicing. An example program is shown in (a) above. The value flow representation for this program produced by one level flow points-to analysis is shown in (b) above.

"10==r" and "10==b" are true, we can logically conclude "r==b" even if we cannot express it as a well-scoped predicate. The predicate slicing algorithm will emit special predicates linking all formal parameters and function return values to an appropriate set of ground terms. These predicates and the transitivity of equality will allow for reasoning that crosses scope boundaries. In the example above, with the predicates "10==r", "5==r", "10==b" and "5==b" we have enough dataflow facts to reason about the value of b after the call to select.

With this intuition in mind, we present the complete algorithm. For each $\mathsf{abortif}(v_i \approx v_j)$:

1. Let $F_i = \mathsf{FlowsInto}(v_i)$.
2. For every $x, y \in F_i$ with $\mathsf{Deref}(x) \rightarrow_F \mathsf{Deref}(y)$, emit the set of predicates $\{$ "a==b" $\mid \mathsf{a} \in \mathsf{Vars}(x) \wedge \mathsf{b} \in \mathsf{Vars}(y) \wedge \mathsf{ShareScope}(\mathsf{a}, \mathsf{b})\}$.[2]
3. Let $T_i = \{\ \mathtt{n} \mid \mathsf{N}(\mathtt{n}) \in F_i\}$ be the set of ground terms n that can flow into v_i.
4. For every $x \in F_i$, consider every $\mathsf{a} \in \mathsf{Vars}(x)$. If $\mathsf{return}(\mathsf{a}) \in C$ or a is a formal parameter then emit the set of predicates $\{$ "n==a" $\mid \mathtt{n} \in T_i\}$.
5. Repeat steps 1–4 with v_j and F_j.

Since the relation $\approx$ is kept abstract, the algorithm generates predicates that keep track of the flow of ground terms throughout the program. Intuitively, the algorithm walks along chains of flow edges and generates equality predicates for every such edge (step 2). Edges that cross scope boundaries are linked using ground terms (step 4). If this set of predicates is used for dataflow analysis, when the $\mathsf{abortif}(\mathsf{v_i} \approx \mathsf{v_j})$ statement is reached some chain of predicates of the form "v_i==x", "x==y", "y==n" should be true. By transitivity, v_i==n. A similar value can be obtained for v_j. Given ground term values for v_i and v_j, the safety of the abortif statement can be decided statically.

For this restricted language the above algorithm can be proved correct: the set of generated predicates is always sufficient to statically verify all $\mathsf{abortif}(\mathsf{v_i} \approx \mathsf{v_j})$ statements. The proof is by induction on the length of chains of flow edges leading into v_i in G and makes use of the fact that in this language v_i must always take on a value from T_i (no other values are possible). The algorithm generates a sufficient number of predicates to keep track of the flow of ground terms through the program.

Example 5 Consider the program in Figure 5(a) and the associated value flow representation in 5(b). Let b be the variable under consideration. In step 1, F_{b} will be $\{10, \mathtt{r}, \mathtt{f}, \mathtt{a}, 5\}$. In step 3 we will emit the predicates "5==a", "f==r", "10==r" (predicates like "a==f" have no valid scope). In step 3, T_{b} is $\{10, 5\}$. So in step 4, f and r qualify so we emit the predicates "5==f", "10==f", "5==r" and "10==r". The algorithm then repeats with c as the variable under consideration and generates "15==c" and "20==c". □

The algorithm can be optimized to produce a smaller set of sufficient predicates if the relation $\approx$ is not abstract (e.g., if it is < or ==) or if one of the

[2] $\mathsf{ShareScope}(\mathsf{a}, \mathsf{b})$ is true if either a or b is a global variable or if a and b are declared in the same procedure.

arguments is constant. For example, to handle abortif($x == 7$), step 3 can be replaced by "Let $T_i = \{7\}$," since at every point we only care if the value that will flow into x is 7 or not. In addition, because value flow representations often merge source variables of different types into the same node, the algorithm can be refined to emit equality predicates over variables of matching types only.

4.5 Experiments

Figure 6 shows the performance of the predicate slicing algorithm on Windows NT device drivers that have been instrumented with input–output request and locking safety properties. In reality, these programs do not fit within the restricted subset of C we presented. For example, the subset does not capture correlations between conditionals: in such cases the algorithm will generate an insufficient set of predicates. As mentioned before, SLAM uses a form of counter-example driven refinement to add in those missing predicates. We compare SLAM with the predicate slicing algorithm against SLAM with all predicates generated by counter-example driven refinement.

"Original Runtime" shows the time for SLAM to either find a bug or prove the driver correct using only counter-example driven refinement. In the case of `iscsiprt`, SLAM does not terminate. "Improved Runtime" shows the execution time when SLAM begins with the predicate slice computed using the previous algorithm. "Generated Predicates" gives the number of distinct predicates in the slice. "Missing Predicates" gives the number of necessary predicates not in the slice that must be found by counter-example driven refinement (e.g., those predicates that correlate conditionals).

Using this technique SLAM was able to scale to some previously unreachable real-world device drivers and performs 2–10 times better on others. The predicate slice provides between all and two-thirds of the necessary predicates. However, since SLAM must generally iterate once to find 3–7 missing predicate and each iteration is exponential in the number of predicates already discovered, the net performance increase is more than linear.

Driver Name	Lines of code	Original Runtime	**Improved Runtime**	Generated Predicates	Max Preds In Scope	Missing Predicates
`apmbatt`	2207	299 s	**22 s**	85	10	0
`pnpmem`	3849	1132 s	**125 s**	143	9	4
`floppy`	7562	1063 s	**600 s**	154	16	33
`iscsiprt`	4543	**	**729 s**	146	10	42

Fig. 6. Performance of predicate slicing. The table above compares the performance of SLAM with predicate slicing against the performance of SLAM with counter-example driven refinement.

5 Related Work

The main idea behind this paper is the use of cheap flow-insensitive value flow information to speed up dataflow. The previous work most similar to our work is that of Ruf [Ruf97] and Zhang et al. [ZRL96]. Ruf showed how a unification-based non-standard type inference procedure such as Steensgaard's pointer analysis could be used to partition the data in a program in such a way that dataflow analysis could be scheduled one partition at a time. This reduces the memory footprint of the dataflow analysis [Ruf97]. Ruf's work can be viewed as producing slices using a points-to graph where all flow edges are replaced by node merging. In general, this will lead to larger slices. Ruf also had no mechanism for identifying constraints and using this information to generate slices of the program.

Zhang et al. used a unification-based pointer analysis to divide the pointer variables in a program into equivalence classes, such that the points-to sets for each equivalence class could be computed separately using a more expensive pointer analysis [ZRL96]. Our work generalizes their result by introducing directional flow, and extends their idea to value flow analysis and clients of value flow analysis in general.

Recent work by Foster et al. [FTA01] uses a unification-based analysis to compute a set of dataflow facts that are then fed to a flow-sensitive type qualifier system. We believe that their work, which appears similar to the typestate slicing used in ESP, could be classified as an instance of our approach.

Rountev et al. developed a framework for combining flow-insensitive global information with flow-sensitive local information [RRL99]. Our work differs from theirs in that our use of flow-insensitive information does not affect the precision of the client analysis, as is the case in their framework. We are merely interesting in improving the efficiency of the subsequent dataflow analysis.

An alternative approach to the one we have presented is to develop demand-driven versions of client dataflow analyses. Previous work on demand-driven dataflow frameworks includes [HRS95] and [DGS95], among others. These frameworks restrict the class of dataflow problems handled, usually to distributive problems, whereas we are interested in value flow problems that are not distributive. It is possible that one could design distributive approximations of value flow analysis that could then be performed from program points of interest, in order to yield more precise slices than those obtained using our method.

Another alternative approach is to use standard program slicing techniques, surveyed in [Tip95]. The drawback of program slicing is that it is based on flow-sensitive reaching definition computation, which is likely to be too expensive to scale to large programs.

The precision of our value flow can be improved through the use of an SSA form [CFR+91].

6 Conclusions

This paper is based on a simple hypothesis: systems that employ heavyweight interprocedural dataflow analyses can benefit greatly by using a inexpensive flow-insensitive value-flow analysis to prune the set of dataflow facts and program statements over which they must operate. We have presented experiments using two different client dataflow analyses to validate our hypothesis. In both cases, we obtain significant gains in performance.

References

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 35, pages 203–213, 2001.

[BR01a] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of SPIN '01, 8th Annual SPIN Workshop on Model Checking of Software*, May 2001.

[BR01b] Thomas Ball and Sriram K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *Proceedings of PASTE '01, ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, June 2001.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CRL99] R. Chatterjee, B. Ryder, and W. Landi. Relevant context inference. In *Proceedings of POPL '99, 26st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI-00)*, 2000.

[DGS95] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Symposium on Principles of Programming Languages*, pages 37–48, 1995.

[DLFR01] Manuvir Das, Ben Liblit, Manuel Fahndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of the 8th International Symposium on Static Analysis*, 2001.

[DLS02] M. Das, S. Lerner, and M. Seigle. ESP: Path sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation (PLDI-02)*, June 2002.

[FTA01] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. Technical Report UCB//CSD-01-1162, University of California, Berkeley, November 2001.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[HRS95] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand-driven interprocedural dataflow analysis. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering Notes*, volume 20, 1995.

[HT01] Nevin Heintze and O. Tardeau. Ultra fast aliasing analysis using CLA: a million lines in a second. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, 2001.

[OJ97] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 1997 International Conference on Software Engineering*, 1997.

[RRL99] Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 235–252, 1999.

[Ruf97] E. Ruf. Partitioning dataflow analyses using types. In *Conference Record of the Twenty-Fourth ACM Symposium on Principles of Programming Languages*, 1997.

[SY86] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[WL95] R. Wilson and Monica Lam. Efficient context-sensitive pointer analysis for C progams. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI-95)*, 1995.

[ZRL96] S. Zhang, B. Ryder, and W. Landi. Program Decomposition for Pointer Aliasing: A Step toward Practical Analyses. In *Fourth Symposium on the Foundations of Software Engineering (FSE4)*, 1996.

Improving the Precision of Equality-Based Dataflow Analyses

Erik Ruf

Microsoft Research
One Microsoft Way
Redmond, WA 90852-6399 USA
erikruf@microsoft.com

Abstract. We present two new, orthogonal techniques for improving the precision of equality-based dataflow analyses. *Subtype expansion* models objects at a per-type granularity, enabling a form of subtype-restricted equality constraint, while *mutation tracking* uses a simple effect analysis to avoid a class of false aliases induced by the bidirectional nature of equality constraints. The utility and costs of these techniques are demonstrated in a context-sensitive interprocedural optimization whose static precision improves by 6-600% when our techniques are applied.

1 Introduction

Equality-based dataflow analyses have become increasingly popular due to their simplicity and efficiency. Such techniques provide flow-insensitive information at a low cost in implementation effort and analysis execution time. Applications include binding time analysis [Hen91], pointer analysis [Ste96b,Ste96a,LH99], escape analysis [Ruf00], call graph construction [GDDC97], synchronization elimination [Ruf00], and thread-specific storage allocation [Ste00].

The benefits of equality-based techniques are offset by a loss of precision with respect to other flow-insensitive techniques such as set-based analysis. A variety of research has addressed this issue. [Ste96a] explicitly models field values within record data structures. Interprocedural context sensitivity has been added both by instantiation of procedure summaries at call sites [LH99,Ruf00] and by deferred traversal of explicit instantiation constraints at query time [FRD00]. [Das00] models assignment with a combination of equality and unidirectional flow constraints, while [DGC98] limits the indegree and outdegree of nodes involved in unidirectional constraints by dynamically switching to equality constraints.

This paper describes two new techniques for achieving additional precision in equality-based systems:

- **Subtype expansion** models individual objects at a per-runtime-type granularity, avoiding overly conservative equality constraints in the face of program type restrictions (static typing, dynamic up- and downcasting, virtual function binding).

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 247–262, 2002.
© Springer-Verlag Berlin Heidelberg 2002

– **Mutation tracking** distinguishes between reads and writes of objects reachable from formal parameters, avoiding the propagation of a class of false (infeasible) aliases from procedure summaries to call sites.

These techniques are largely orthogonal to one another and to the prior work referenced above. We demonstrate this by describing and assessing our techniques in the context of a system whose base interprocedural framework configuration performs field modeling and procedure summary instantiation. For a sample optimization that removes unnecessary downcast checks, the static and dynamic fractions of total checks removed are improved by 6-600% and 0-4400%, respectively.

2 Background

The techniques described in this paper are implemented in the Marmot native compilation system for Java [FKR+00]. Marmot implements most Java 1.1 semantics and libraries [GJS96], but does not support dynamic loading and limits the use of reflection. The resulting "closed-world" assumption enables a number of whole-program analyses, including an equality-based, flow-insensitive, context-sensitive analysis with method specialization [Ruf00]. This section briefly summarizes relevant aspects of this platform, which we will call the *base analysis*.

Runtime values of reference type (objects) are modeled by a union-find data structure called an *object representative*[1] that contains a list of attributes (arbitrary elements of dataflow lattices) and a mapping from field names to object representatives that represent field and array element values. Some attributes, such as *static* (indicating that the modeled runtime values may be transitively reachable from a static field), are recursively applied to an object reference's fields, while others, such as *synchronized* (indicating that an object may be the target of monitor lock/unlock operations) are not.

Intraprocedural analysis consists of traversing a method body in the context of a local symbol table that maps formal and local variables to new object representatives. The traversal evaluates expressions (which may create, dereference, or add attributes to object representatives), assignments (unifying [ASU86] the object representatives for the left and right hand sides), and method invocations. Nonrecursive invocations are modeled by a *mapping* algorithm similar to type instantiation (see Figure 1) that propagates aliases and attributes from the target method summary to the call site. Recursive invocations simply unify formal/actual, return/result, and thrown/caught representatives.[2]

Interprocedural analysis performs two walks of a topologically sorted strongly connected component (SCC) decomposition of a conservative static call graph.

[1] Object representatives were called *alias sets* in [Ruf00].

[2] Optionally, we could obtain additional precision by also using the mapping algorithm at recursive call sites. Doing so would require iteration over such sites in each call-graph SCC under an extended-occurs-check predicate [Hen93]. Our preliminary experiments with such an approach have experienced scalability issues in the analysis of large SCCs.

```
mapValues(summary, site) {
  memoTable = new MemoTable
  for (i=0; i<|summary|; i++)
    map(summary[i], site[i], memoTable)
}
map(repSum, repSite, memoTable) {
  if (repSum.isStatic)
    unify(repSum, repSite)
  else if (memoTable.containsKey(repSum))
    existing = memoTable[repSum]
    unify(existing, repSite)
  else
    memoTable[repSum] = repSite
    repSite.addAttribs(repSum.attribs)
    foreach (fieldname fn in repSum.fieldMap.keys)
      if (repSite.fieldMap[fn]==null)
        repSite.fieldMap[fn] = new ObjRep
      map(repSum.fieldMap[fn], repSite.fieldMap[fn], memoTable)
}
```

Fig. 1. Pseudocode for mapping procedure

In the upward (callee→caller) propagation phase, the intraprocedural analysis is applied to each method in the current SCC, after which a signature, or *method summary* is extracted from each method's symbol table. An optional *pruning* operation compresses each summary by removing object representatives that cannot induce aliasing or attribute propagation at call sites. No fixed point iteration is required because recursively dependent summaries share object representatives.

The downward (caller→callee) interprocedural analysis is similar, except that the direction of propagation is reversed, and only attributes, not aliases, are propagated. After all of a method's call sites are processed, its dataflow information is complete, enabling transformation. Some versions of our analysis also perform *method specialization*, in which method signatures and bodies are copied to enable call-site-specific transformations.

Because the mapping procedure does not employ k-limiting, the analysis could consume exponential space and time with respect to program size. However, given that few large recursive data structures are constructed without iteration or recursion, neither of which generate additional object representatives, this is unlikely in practice.

3 Subtype Expansion

3.1 Motivation

Our first technique focuses on the granularity at which the analysis models objects. In the base system, runtime objects are represented by static object representatives consisting of list of attributes describing the object and a mapping from fieldnames to object representatives. All forms of merging, including local assignment, exception dispatch, and the interprocedural value mapping that occurs at call sites, are implemented via unification[3] of these object representatives. Similarly, all forms of attribute assignment, such as marking an object as escaping or as having a particular runtime type, affect the attribute list(s) of their argument object representative(s).

This monolithic representation becomes overly conservative when subtyping is introduced. In an object-oriented language, both explicit (e.g., downcasting, `instanceof`) and implicit (e.g., virtual function and exception dispatch) operations can restrict the runtime types of objects. This restriction can be used to limit the scope of merging or attribute assignment involving such objects.

```
static Object myToString(Object o) {
  S1: return o.toString();
}
```

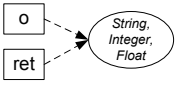


Fig. 2. Virtual call example. The base analysis infers an alias between `o` and the value returned by `S1`.

```
Object G = ... //escapes
Object x = ...
if (...)
  S2: G.f = (T1)x;
else
  Object y = (T2)x;
```

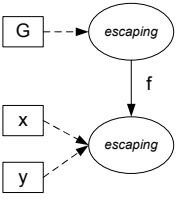


Fig. 3. Escape example. The base analysis marks both `x` and `y` as escaping.

Figure 2 shows a simplified version of a case that arose in [Ruf00]. During the process of constructing a summary for method `myToString`, a virtual call

[3] Our implementation performs unification and/or mapping as the merging operations are processed. However, our exposition applies equally well to systems that construct explicit constraints to be solved later.

with selector `Object.toString` and a receiver argument of static type `Object` is detected. Given that the argument's dynamic type is bounded only by the type `Object`, all known summaries of `toString` are applied. Most such applications are benign; e.g., `Integer.toString` and `Float.toString` merely return new string objects that point to new character arrays. However, `String.toString` is specified [GJS96] to return its receiver argument (no copying allowed), so applying it causes the object representatives for argument `o` and return value `r` to be unified, thus aliasing the receiver and result values, even when the receiver is not a `String`. This would make it impossible, for example, to remove the dynamic type check in the expression `(String)myToString(o)`, even though `myToString` can only return strings.

Similar issues can arise in purely intraprocedural scenarios, and in analyses whose purpose is other than type propagation. For example, in Figure 3, if types `T1` and `T2` are unrelated in the class hierarchy, the fact that `(T1)x` escapes should not cause `y` to escape, but (under the base analysis) it does.

3.2 Solution

The common theme in the above examples is that multiple runtime objects described by a single analysis-time object representative may reach different operations depending on their [the objects'] runtime types. For example, only string objects `o` will cause an invocation of `String.toString` at (Fig. 2, `S1`), and only objects of type T1 will escape at (Fig. 3, `S2`).

Subtype expansion models this behavior by changing the granularity at which runtime objects are statically modeled, allowing an object representative to expose multiple union-find data structures.

In one maximally-general implementation, each top-level object representative carries a vector of at most $|\mathcal{T}|$ sub-representatives, where $\mathcal{T}$ is the set of all reference types in the program. Each sub-representative models the object's behavior (i.e., carries attributes and a field mapping) at a set of types $\mathcal{T}_i \in \mathcal{T}$ where all $\mathcal{T}_i$ are disjoint and $\bigcup_i \mathcal{T}_i = \mathcal{T}$.[4] Unification of object representatives o_1 and o_2 "at" a type $T \in \mathcal{T}_i$ consists of unifying the sub-representatives $o_1.[\mathcal{T}_i]$ and $o_2.[\mathcal{T}_i]$as well as all pairs of object references $(r_1.\mathit{fields}[f], r_2.\mathit{fields}[f])$ where T declares or inherits a field f. Such a representation allows for arbitrary type-based restriction of value flow. For example, it can precisely represent $\{t \mid t \not\leq T\}$, which could be imposed on the type of `y` in a `catch` clause guarding `x = (T)y`.

Our prototype implementation achieves a lower space cost by precisely modeling only restrictions to subtype hierarchies (type ideals) rather than arbitrary sets of types. Our representation mirrors the the underlying type hierarchy. Each object representative is associated with a single reference type R, and carries an

[4] The partitioning of $\mathcal{T}$ into subsets $\mathcal{T}_i$ can take place statically or dynamically. In the static case, maximal precision is obtained by choosing $\mathcal{T}_i$ of size 1. In the dynamic case, precision is maintained by splitting existing subrepresentatives when a finer-grained restriction is required.

attribute list, a field mapping, and a "subtype" mapping (from immediate subtypes of R to object representatives). Field mappings in "subtype" representatives share state (mappings for inherited fields) with those of their supertypes.[5] Each runtime value is modeled by an object representative whose associated type is `Object`. Unifying two such representatives at a type T consists of unifying the subtype representatives associated with T, which will induce pairwise unification of representatives of fields directly implemented by types $T' \geq T$.

The implementation of attributes under subtype expansion differs slightly from that under the base analysis (as described in Section 2). Transitive reachability attributes such as *static* now recurse over both field and subtype representatives, while object-level attributes such as *synchronized* recurse over subtype representatives. A third variety of attribute that applies only to a single object representative can be used to represent information that is relevant only at a particular type such as a boolean flag that models whether the represented object is allocated at a particular type.

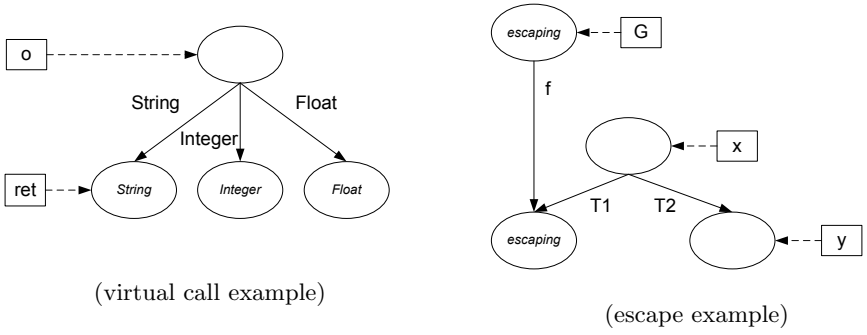


Fig. 4. Example representations with subtype expansion

Figure 4 shows the representation of the example programs in Figures 2 and 3 when subtype expansion is used. In the virtual call example, only string-valued instances of `o` can be returned, even though `o`'s static type is `Object`. In the escape analysis example, only subtypes of `T1` escape, despite `x`'s declaration at type `Object`.

The primary costs of subtype expansion arise from the increased size of object representatives and the increased number of unification operations required. In the asymptotic worst case, these may be multiplied by the size of the class hierarchy and (in our implementation) the maximum number of fields visible to a class.

[5] This implementation can be viewed as being "factored" with respect to subtypes, but not with respect to fields. A more efficient representation might avoid this duplication; e.g., by making field maps first-class, unifiable datatypes and allowing a subtype representative to reference its parent's field map.

4 Mutation Tracking

4.1 Motivation

Our second technique addresses the bidirectional nature of equality constraints. Within procedures, a typical equality-based analysis ensures that any two object references that could represent the same runtime value constrained to be equal. Interprocedurally, it ensures that call-site object references (and their attributes) are instances of the corresponding references in the callee. This can result in unwanted aliasing.

Consider a call to procedure `p`, whose code and method summary are shown in Figure 5. Under our base analysis, the alias created in statement `S1` is reflected at the call site by making `a2`'s representative reachable from that of `a1` (via field f). Similarly, `S3` causes the object reference associated with `a5` to be unified with that of `r`. These operations are sufficient to represent (at the call site) the aliasing effects of calling `p`. However, the analysis also unifies the representatives of `f3.g` and `f4.h` because they appear in a common context, namely `S2:x`, thus ensuring the existence of a common representative for `a3.g` and `a4.g`, marked with the "synchronized" attribute. In fact, only the attribute need be propagated—the call does not induce an alias between `a3.g` and `a4.h`.

A more complex situation arises with container objects such as `Vector` and `Hashtable`, instances of which could contain themselves, thus causing the representative for `this` to be unified with those of the elements of the array(s) used to implement the container. For example, invocations of `Hashtable.get` and `Hashtable.equals` cause the analysis to infer caller-side aliases between the table, its keys, and its values, even though neither procedure mutates the table.

```
p (f1, f2, f3, f4, f5) {
  S1: f1.f = f2;
  S2: x = (...) ? f3.g : f4.h;
  S3: synchronized(x) { ... }
  S4: return f5
}
r = p(a1, a2, a3, a4, a5);
```

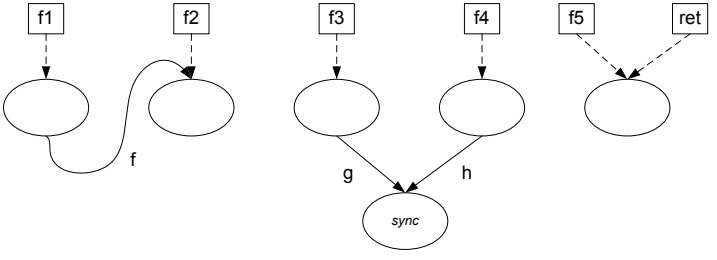


Fig. 5. False alias example

4.2 Solution

Mutation tracking avoids false actual/actual aliases by distinguishing between two kinds of equality constraints. Some constraints, such as that inferred for statement `S1` in Figure 5, add a potentially new path (alias) from one caller-side object to another. For correctness, such constraints must be reflected to the call site when the method summary is applied. Other equality constraints arise merely because multiple formal-reachable representatives reach a common context (e.g., Figure 5, statement `S2`). While such constraints are necessary for the propagation of attributes, they do not induce any new paths between caller objects and thus should not induce caller-side constraints.

Our implementation of mutation tracking distinguishes these cases by adding an optional *mutation* attribute to the object representative data structure. Mutation attributes are recursively propagated across child fields and (if subtype expansion is used) child types. Whenever the intraprocedural analysis encounters a field or array element update, it sets the mutation attribute of the representative for the new value, representing the fact that the new path(s) represented by the resulting unification operations arose due to mutation rather than context sharing.

Mutation attributes are used to limit alias propagation at call sites. This requires changes to the processing of procedure invocations. At nonrecursive call sites, the mapping process of Figure 1 is changes as follows: a callee representative f is added to the memo table only when it

1. is reachable from the return or thrown representative, or
2. transitively reaches a representative having the mutation attribute

Note that even when a representative is not added to the memo table, its attributes are still propagated.

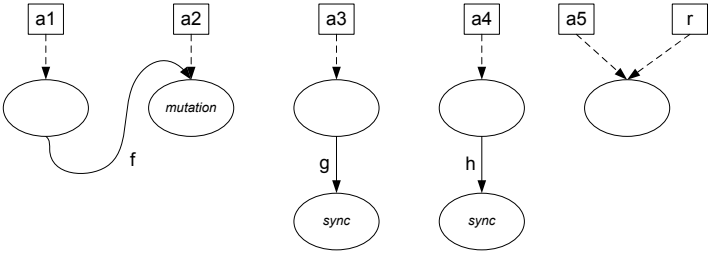


Fig. 6. False alias example with mutation tracking

In the example of Figure 5, the representative shared by `f1.f` and `f2` will have the mutation bit set, forcing unification of `a1.f` and `a2`, with the result having the mutation attribute. The representatives for `r` and `a5` are unified because `f5`'s representative is returned. The representative for `x` is neither returned nor

marked as a mutation, As shown in Figure 6, the representatives for `a3.g` and `a4.g` are both marked with the synchronized attribute, but are not unified.

The treatment of recursive calls need not be changed; unifying corresponding caller and callee object representatives as before is sufficient. Doing so preserves the property that no fixed point iteration is required, but may result in the propagation of some false aliases within strongly connected components of the call graph.

Alternatively, the analysis can use the same criteria for justifying unification at recursive calls as is used for justifying memo table insertion at nonrecursive calls. However, unlike the nonrecursive case, here the algorithm must make its decision with incomplete knowledge, as the analysis of the callee procedure may be incomplete. This requires iteration over all recursive calls in the current SCC until no more call-site unifications can be performed. We have implemented and assessed both of the above techniques.

For nonrecursive calls, the additional tests (1) and (2) above can either be computed in constant time or are already performed as part of the pruning process. In the case of recursive calls, item (2) requires an additional search of the formal representatives. If required, iteration only reprocesses formal/actual pairs where unification has been deferred.

5 Assessment

5.1 Sample Analysis

We tested our improvements on a simple prototype analysis that models runtime types of objects. Type propagation is an interesting application for equality-based techniques because (unlike the more common application, escape analysis) it is a unidirectional problem, in which the bidirectional nature of equality-based techniques leads to additional imprecision (often called "backflow").

For the sake of simplicity, we use the analysis to transform only explicit downcast operations, and do not perform specialization on the downward pass. All versions of the analysis make use of the intermediate representation described in Section 3.2.[6] On the upward pass, constants and objects returned by constructors carry their runtime type in the form of an *allocated* annotation on the object representative for the appropriate subtype of `Object`. On the downward pass, each explicit downcast test (on type T) is checked: if all allocated annotations on the operand's (transitive) "subtype" children are on object references for types $T' \leq T$, the test is removed.

Table 1. Benchmark programs

name	meths	stmts	checks	
			static	dynamic
_213_javac	2091	33319	321	6.41E+06
_202_jess	1541	20703	95	1.51E+07
_209_db	799	9268	42	5.32E+07
_228_jack	1043	17825	122	5.70E+06
jlex100	637	11413	67	1.14E+08
javacup	948	17786	434	1.49E+06

5.2 Benchmarks

We tested our techniques on six programs (described in Figure 1) chosen because they either perform a large number of downcast check operations or make extensive use of object-polymorphic container types.

The analysis and optimization were implemented in the Marmot [FKR+00] optimization pipeline immediately following initial parsing and cleanup, but before inlining, local type propagation, etc. Marmot is a whole-program compiler; the optimization was applied to the user program and most non-native library methods (a few methods involving low-level synchronization, etc. were explicitly modeled and left untransformed).

We tested six distinct configurations of the prototype. The "base" configuration enables the analysis without without extensions. The remaining configurations selectively enable combinations of three techniques: "s" denotes subtype expansion, "m" denotes mutation tracking, and "f" denotes fixed point iteration.

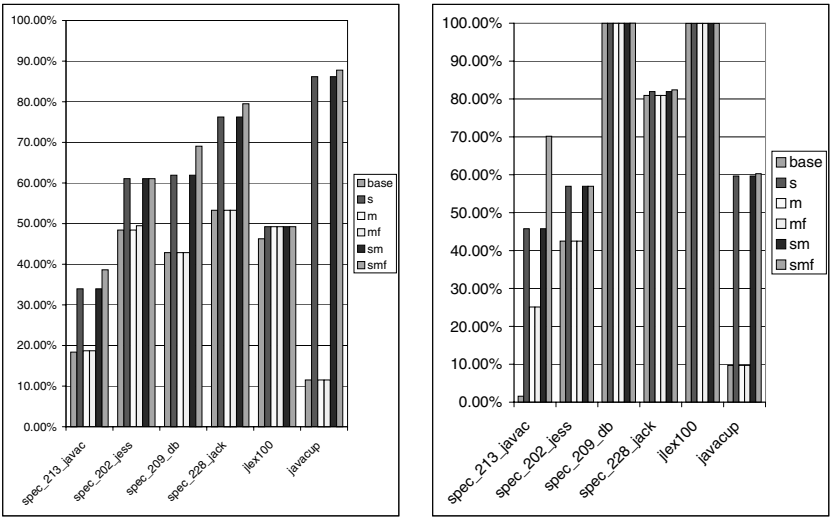
All testing was performed on a 1.7GHZ dual-processor Intel Xeon processor with 1GB of RAM under Windows XP Professional, using the Marmot generational copying collector with a 256MB heap. All of the benchmark programs executed deterministically, so all operation counts are precise; timings are the average of multiple executions.

5.3 Precision

Figure 7 shows the fraction of static and dynamic downcast checks for each benchmark program and configuration.[7] With the exception of `jlex100`, our techniques removed at least 10% more static checks than the base analysis, with subtype expansion responsible for the majority of the gain. Mutation tracking improved upon the base analysis slightly on `javac` and `jlex100`; iterated mutation tracking improved `jess`. Adding mutation tracking to subtype expansion

[6] This choice improves the precision of our assessment by avoiding accidental differences introduced by representation changes, but does exact a price in analysis time for versions not making use of the subtype expansion capability.

[7] Absolute data are available in the Appendix.



(static) (dynamic)

Fig. 7. Fraction of cast check operations removed

had little effect unless iteration was enabled, where it had significant effects on four of six programs.

The improved static precision had little to no dynamic effect in the programs `db`, `jack`, and `jlex`, where the base analysis already achieved excellent results. In the remaining programs, substantial improvements were obtained, with an additional 15-69% of dynamic checks removed.

Given that the sample analysis emphasized type precision rather than alias effects (which would be more important in, e.g., an escape analysis), it is not entirely surprising that most of the measured improvement came from subtype expansion. More interesting, perhaps, is that mutation tracking exposed additional opportunities only when iteration was used, suggesting that false aliases at potentially recursive call sites are more problematic than those at nonrecursive sites. This effect may be due to the use of dynamic object-based polymorphism to implement homogeneous containers (vectors and hash tables) where static parametric polymorphism would suffice.

5.4 Execution Times

The primary purpose of this study was to evaluate the precision improvements available via our techniques, not their value in optimization. Nonetheless, we did capture execution times, and found that they were reduced by 1-20%, but the degree of improvement was not well correlated with the dynamic removal counts. This information, combined with the knowledge that Marmot's checks are quite

efficient, suggest that other effects, such as altered instruction schedules and register pressure changes, may be responsible for much of the improvement.

5.5 Analysis Times

Table 2 gives relative analysis time costs for the various configurations of our analysis. For the most part, analysis times grew by less than 60%, but some outlying values did raise concerns:

- The additional cost of subtype expansion strongly depends on the representation used. Our prototype representation (see Section 3.2) factors attributes, but requires explicit instantiation of shared field representatives when a type restriction occurs. This is problematic for programs having many fields of reference type, and would need to be addressed in any serious implementation.
- For `javac` and `jess`, the cost of mutation tracking with fixed point iteration is significant. Given that relatively few call sites are visited multiple times during iteration, we suspect that much of the cost lies in searching callee-side signatures to determine if a unification operation is required.

Table 2. Relative analysis times (base=1)

name	s	m	mf	sm	smf
`spec_213_javac`	5.43	1.01	13.73	5.84	22.33
`spec_202_jess`	1.16	1.06	5.67	1.23	3.17
`spec_209_db`	0.96	1.11	1.37	1.02	1.00
`spec_228_jack`	0.38	1.04	1.58	0.39	0.47
`jlex100`	1.07	1.04	1.15	1.07	1.15
`javacup`	0.96	1.03	1.55	0.99	1.09

6 Related Work

6.1 Precision of Equality-Based Analyses

Precision improvements for equality-based analyses can be broadly grouped into two categories. One group of techniques refines the granularity at which individual objects are modeled (e.g., by adding equality representatives for individual fields [Ste96a]). This finer granularity allows for a larger set of equivalence classes, some of which may refer only to some "aspects" of an object. Subtype expansion belongs to this category.

Another group of techniques relies on avoiding equalities where possible. Summary based context sensitivity does this at call sites, where equality in the

callee need not induce equality in the caller. Mutation tracking is merely a way to avoid a specific class of such propagation. Other variants combine equality and flow constraints. Das's OLF technique[Das00] is of particular relevance in that it preserves flow directionality only for top-level interactions between objects, and requires equality for "content" properties such as pointer targets. Our techniques lack the precision of flow constraints, but do have the advantage of applying equally to all aspects of an object. Combining our techniques with OLF might yield interesting results.

6.2 Other Context-Sensitive Analyses

A number of non-equality-based, context-sensitive flow analyses have been developed in recent years. These systems either construct and explicitly instantiate method summaries [CGS+99,WR99,LH99,CRL99,Ruf00] to avoid pollution from multiple call sites or add instantiation constraints and flow polarities to enable context-sensitive queries after the initial analysis is complete [FRD00, RF01]. In both cases, adding subtype expansion should be straightforward, as it only requires changes to the object representation and not to the propagation of constraints. Mutation tracking, on the other hand, adds a new variety of intraprocedural information that guides constraint propagation at call sites, and thus might require additional modification.

7 Conclusion

We have presented two new, orthogonal techniques for improving the precision of equality-based dataflow analyses:

- **Subtype expansion** extends the static object representation to model equality between objects viewed at a particular shared subtype (or subtype ideal). This is similar to the extension of simple equality models with explicit representatives (types) for object fields, but operates in an orthogonal dimension. The extension of unification and context-sensitive call mapping to the expanded representation is straightforward, suggesting that systems using flow dependences could be similarly extended. Practical application of subtype extension will require additional experimentation with precision/complexity tradeoffs in the representation of subtype-specific information.
- **Mutation tracking** explicitly models the mutation of state reachable from formal parameters, avoiding a class of false aliases induced by the bidirectional nature of equality constraints. This technique applies primarily to summary-based context sensitivity techniques where the absence of mutation can be used to avoid the propagation of an alias or other dependency from a function to its caller(s). When combined with fixed point iteration, mutation tracking can provide a coarse-granularity form of flow directionality at recursive call sites.

Acknowledgements. We thank the ex-members of the Advanced Programming Languages group at Microsoft Research for developing the Marmot compiler infrastructure, and Todd Knoblock, Bjarne Steensgaard, and Manuvir Das for interesting and relevant conversations.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, USA, 1986.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, November 1999.

[CRL99] Ramkrishna Chatterjee, Barbara G. Rynder, and William A. Landi. Relevant context inference. In *Proceedings 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–146, January 1999.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[DGC98] Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *Proceedings 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 222–236, 1998.

[FKR+00] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for Java. *Software: Practice and Experience*, 30(3):199–232, March 2000.

[FRD00] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '97)*, pages 108–124, October 1997.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.

[Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In *Functional Programming and Computer Architecture*, pages 448–472, 1991.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[LH99] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings FSE'99*, pages 199–215. ACM, 1999.

[RF01] Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: from polymorhphic subtyping to CFL-reachability. In *Proceedings 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2001.

[Ruf00] Erik Ruf. Effective synchronization removal for Java. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 208–218, June 2000.

[Ste96a] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *International Conference on Compiler Construction*, number 1060 in Lecture Notes in Computer Science, pages 136–150, Where?, April 1996.

[Ste96b] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, FL, January 1996.

[Ste00] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In *Proceedings of the ISMM 2000 International Symposium on Memory Management*, July 2000. also published as SIGPLAN Notices 36(1), January 2001, ACM.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, November 1999.

Appendix

Table 3. Raw benchmark results

(a) Static check operations

name	unopt	base	s	m	mf	sm	smf
spec_213_javac	321	262	212	261	261	212	197
spec_202_jess	95	49	37	49	48	37	37
spec_209_db	42	24	16	24	24	16	13
spec_228_jack	122	57	29	57	57	29	25
jlex100	67	36	34	34	34	34	34
javacup	434	384	60	384	384	60	53

(b) Dynamic check operations

name	unopt	base	s	m	mf	sm	smf
spec_213_javac	6408484	6308416	3476204	4799096	4799096	3476204	1910732
spec_202_jess	15071248	8666204	6485130	8666204	8666204	6485130	6485130
spec_209_db	53229831	19047	19047	19047	19047	19047	0
spec_228_jack	5703096	1086440	1028755	1086440	1086440	1028755	1002762
jlex100	113790000	61200	47000	47000	47000	47000	47000
javacup	1485504	1341350	598912	1341350	1341350	598912	589789

(c) Analysis times (sec)

name	base	s	m	mf	sm	smf
spec_213_javac	5.84	31.70	5.92	80.26	34.14	130.47
spec_202_jess	2.77	3.20	2.92	15.69	3.41	8.77
spec_209_db	0.88	0.84	0.97	1.20	0.89	0.88
spec_228_jack	4.56	1.72	4.75	7.22	1.78	2.16
jlex100	0.42	0.45	0.44	0.48	0.45	0.48
javacup	1.08	1.03	1.11	1.67	1.06	1.17

(d) Execution times (sec)

name	unopt	base	s	m	mf	sm	smf
spec_213_javac	2.843	2.843	2.780	2.796	2.828	2.812	2.812
spec_202_jess	2.202	2.171	2.139	2.171	2.171	2.155	2.186
spec_209_db	10.186	9.296	9.296	9.264	9.312	9.327	9.280
spec_228_jack	0.983	0.921	0.921	0.952	0.936	0.921	0.906
jlex100	3.781	3.014	3.046	3.124	3.093	2.999	3.015
javacup	0.189	0.186	0.184	0.183	0.186	0.180	0.180

Guaranteed Optimization: Proving Nullspace Properties of Compilers

Todd L. Veldhuizen and Andrew Lumsdaine

Indiana University, Bloomington IN 47401 USA
tveldhui@acm.org, lums@osl.iu.edu

Abstract. Writing performance-critical programs can be frustrating because optimizing compilers for imperative languages tend to be unpredictable. For a subset of optimizations – those that simplify rather than reorder code – it would be useful to prove that a compiler reliably performs optimizations. We show that adopting a "superanalysis" approach to optimization enables such a proof. By analogy with linear algebra, we define the *nullspace* of an optimizer as those programs it reduces to the empty program. To span the nullspace, we define rewrite rules that de-optimize programs by introducing abstraction. For a model compiler we prove that any sequence of de-optimizing rewrite rule applications is undone by the optimizer. Thus, we are able to give programmers a clear mental model of what simplifications the compiler is guaranteed to perform, and make progress on the problem of "abstraction penalty" in imperative languages.

1 Introduction

In our experience developing high-performance numerical libraries for object-oriented languages [17,14] we have found that industrial compilers fail to eliminate abstraction reliably. In fact, the best optimizing compilers, while achieving remarkably good performance, are capricious, achieving that performance only when some unpredictable phrasing of the program is presented. This paper reports on progress in devising a compiler structure that achieves guaranteed optimization for imperative languages.

Optimizing compilers map programs to programs. It is interesting to ask as one does in linear algebra: what is the nullspace (or kernel) of the mapping? Clearly optimizing compilers are not linear transforms, but the analogy is useful. The nullspace of a linear transform A is the set of vectors transformed to the zero vector: $null(A) = \{x \mid Ax = 0\}$. For compilers, a sensible choice of nullspace is the set of programs reduced by the optimizer to the empty program $\emptyset$. By $\emptyset$, we mean the syntactically minimal program generating the value 0; in C, it is the program `int main() { return 0; }`. We use the notation $\mathcal{O}[\![\cdot]\!] : \mathsf{Program} \rightarrow \mathsf{Program}$ for an optimizer, and define its nullspace to be $null(\mathcal{O}) = \{p \in \mathsf{Program} \mid \mathcal{O}[\![p]\!] = \emptyset\}$.

Program code may be loosely divided in two: "useful" code that implements desired semantics of the program, and "abstraction" code that does not change observable properties but is introduced to further software engineering goals such

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 263–277, 2002.
© Springer-Verlag Berlin Heidelberg 2002

as encapsulation and modularity. For example, we might want our program to compute:

$$1 + 2$$

but for software engineering reasons we write:

```
x = new Integer(1);
y = new Integer(2);
plus(x, y)
```

The resulting performance hit is often called the *abstraction penalty* [15,11,10] and is a pressing concern for those attempting to meld good software engineering practice with high performance. Ideally, optimizing compilers would eliminate abstraction while preserving semantics. In other words, we would like the nullspace of compilers to encompass typical patterns of software engineering abstraction.

Carrying the analogy further, in linear algebra we have the property that if A is a linear transform, then $z \in \mathit{null}(A)$ implies $A(x + z) = Ax$; that is, adding something from the nullspace has no effect. How does one account for "adding" abstraction to a program? Here the analogy to linear algebra is weak, and we turn to rewrite systems.

Term rewriting systems have a long history of use in program optimization (e.g. [19,4,20]). We consider an application opposite to their usual use: as rules that de-optimize a program, introducing abstraction, rather than optimizing it. Here are two such rules, stated informally:

1. Replace any expression e with x, where x is a fresh variable, and insert x $:= e$ immediately before e.
2. Select a subset of local variables, allocate a record of sufficient size at an appropriate program point,[1] replace variable definitions with writes and variable uses with reads to appropriate slots of the record.

The above rules span rudimentary OOP-style encapsulation: applying them to "1" and "2", we can produce code resembling x = new Integer(1); y = new Integer(2).

We regard such de-optimizing rules as a basis for the desired nullspace; if p is a program, $\mathcal{O}[\![\cdot]\!]$ the optimizer, and $\{g^i\}$ are de-optimizing rules, we want $\mathcal{O}[\![g^k g^{k-1} \cdots g^1 p]\!] = \mathcal{O}[\![p]\!]$ for any sequence of rule applications $g^k g^{k-1} \cdots g^1$. This is our analogy to "additivity" in linear algebra.

To build an optimizer capable of undoing any application of such rules, it is necessary to avoid *phase ordering problems* – problems that arise when optimizers are constructed from discrete optimizing passes. We do this by adopting a *superanalysis* approach in which all analyses are performed simultaneously,

[1] To be exact: a program point that dominates all definitions and uses of the variables.

followed by a single transformation step. This approach is known to be more powerful than applying individual optimizations in sequence (even iteratively), since optimizations are synergistic ([22,1]). It also enables a proof of the nullspace property: any sequence of rule applications is undone *by a single application of the optimizer*. The nullspace proof is an induction over rule applications: for each rule we consider the changes made to the analysis equations, show the resulting system of equations has a "consistent" solution, and that the transformation step erases the code added by the rule. We demonstrate the proof technique by proving the nullspace property for a small arithmetic language. In an accompanying technical report, we prove the nullspace property for a model optimizer of an imperative language; we summarize the results here.

This paper is not about a proof for proof's sake. Rather, the proof guides the design of the optimizer. To enable the nullspace proof one is obliged to make many changes to the structure of the optimizer; and once the proof is complete, one has an optimizer design with the desired property.

1.1 Related Work and Contributions

The notion of guaranteed optimization is well-known in the functional world. Compilers for functional languages often guarantee tail-call optimization; staged languages such as MetaML [16] guarantee that expressions annotated as static will be evaluated fully at compile time; deforestation [21] can reliably eliminate intermediate representations for certain expressions [6]. Sands [13] in his work on improvement theory shows that certain optimization algorithms offer strong guarantees. The use of rewrite rules for optimization can guarantee (in restricted cases) the existence of normal forms [5].

In the imperative world, the vagaries of effects, unruly control flow, and absence of the algebraic properties of pure functional languages make guaranteed optimization difficult. The contribution of our work is to propose the idea of a nullspace as a guiding principle for guaranteed optimization of imperative languages, and to introduce a proof technique for proving nullspace properties of compilers. We believe this to be a first step toward a practical theory of minimal normal forms for imperative programs.

2 Definitions

We adopt abstract interpretation-style lattices [2] in which $\bot$ is associated with "absent" and $\top$ with "conflicting information." This is opposite from the convention in data flow analysis. We use $\sqsubseteq$ for a partial order relation, $\sqcup$ for lattice join, and $\times$ for lattice direct product. A tuple of lattices $(D_1, \ldots, D_n)$ is understood to be the lattice $D_1 \times \ldots \times D_n$. We reserve $\wedge$ to mean logical conjunction.

Program analyses are often presented as special-purpose solvers that simultaneously build and solve lattice equations. To reason about analyses, we take the view that a program analysis builds a system of equations such as:

$$\begin{array}{l} x_1 = e_1(x_1, x_2, \ldots, x_n) \\ x_2 = e_2(x_1, x_2, \ldots, x_n) \\ \vdots \qquad\qquad\qquad \vdots \\ x_n = e_n(x_1, x_2, \ldots, x_n) \end{array}$$

where the x_i are analysis variables and the e_i are monotone functions. We write $X = E(X)$ for the above system, where $X = (x_1, \ldots, x_n)$ and $E = (e_1, \ldots, e_n)$. We use the notation lfp E to mean the least fixpoint of the function E.

Definition 1. *A system of lattice equations S is a pair $S = (D, E)$ where $D = (D_1, \ldots, D_n)$ is a tuple of complete lattices and $E = (e_1, \ldots, e_n)$ is a tuple of monotone functions $e_i : D_1 \times \ldots \times D_n \to D_i$. For variables $X = (x_1, \ldots, x_n)$ with $x_i : D_i$, we define $X^\star(S) = (x_1^\star(S), \ldots, x_n^\star(S)) = \text{lfp } E$ to be the solution (least fixpoint) of the system of equations $X = E(X)$.*

We will write $X^\star$ to mean $X^\star(S)$, and similarly for other parameters when they are apparent from context.

Definition 2. *An optimizer $\mathcal{O}$ is a pair $(\mathcal{A}, \mathcal{T})$ where:*

- *$\mathcal{A}$ is an analysis taking a program p and producing a system of lattice equations $S = (D, E)$ and a mapping M between program points and elements of E;*
- *$\mathcal{T}$ is a transformation taking a solution $X^\star(S)$, program p, mapping M, and producing a new program p'.*

and we write $\mathcal{O}[\![p]\!] = p'$.

3 Example: Optimization of Arithmetic Expressions

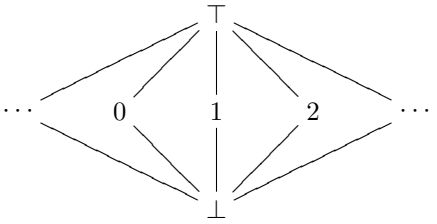
To introduce the proof technique we consider optimizing arithmetic expressions comprised of integers (n), variables (x) and addition:

$$e ::= n \mid \mathsf{x} \mid \ +(e, e)$$

We study two "de-optimizing" rewrite rules:

$$\begin{array}{l} g_1 : n \mapsto +(n_1, n_2) \quad \textit{where } n_1 + n_2 = n \\ g_2 : e \mapsto +(e, 0) \end{array}$$

For example: $\mathsf{a} \overset{g_2}{\mapsto} +(\mathsf{a}, 0) \overset{g_1}{\mapsto} +(\mathsf{a}, +(4, -4)) \overset{g_1}{\mapsto} +(\mathsf{a}, +(+(1, 3), -4))$. We will define an optimizer, then prove that it undoes any sequence of applications of g_1 and g_2 in a single step. This is not an impressive result, but serves to introduce the proof technique. For this simple example, a single analysis suffices – constant propagation. We use the lattice:



The analysis $\mathcal{A}$ takes an expression and constructs (1) a system of analysis equations; and (2) a mapping between analysis variables and program points. We represent the mapping by annotating an expression with analysis variables, such as ${}^{c_0}+({}^{c_1}1, {}^{c_2}2)$. The analysis rules are:

Expression	**Equation**
${}^{c_0}n$	$c_0 = n$
${}^{c_0}\mathsf{x}$	$c_0 = \top$
${}^{c_0}+({}^{c_1}e_1, {}^{c_2}e_2)$	$c_0 = \hat{+}(c_1, c_2)$

where the abstract version of $\hat{+}$ is:

$$\hat{+}(x,y) \equiv \begin{cases} \bot & \textit{if } x = \bot \textit{ or } y = \bot; \textit{ else} \\ x + y & \textit{if } x \in \mathbb{Z} \textit{ and } y \in \mathbb{Z}; \textit{ else} \\ \top & \end{cases} \tag{1}$$

A key requirement of our proof technique is that the analysis be *compositional*: the system of analysis equations for an expression is obtained by composing analyses of subexpressions. This implies that a rewrite on an expression induces a rewrite on the analysis equations in a straightforward way. Consider the single rewrite:

$$+(\mathsf{a}, 0) \overset{g_1}{\mapsto} +(\mathsf{a}, +(4, -4))$$

It is convenient to think of the rewrite in terms of its context and redex: in the above example we have the context $C = +(\mathsf{a}, [\ ])$ and the redex 0, where $[\ ]$ denotes a hole. The analysis of the two expressions is:

Mapping	${}^{c_0}+({}^{c_1}\mathsf{a}, {}^{c_2}0)$	${}^{c_0}+({}^{c_1}\mathsf{a}, {}^{c_2}+({}^{c_3}4, {}^{c_4}-4))$
Equations	$c_0 = \hat{+}(c_1, c_2)$ $c_1 = \top$ $c_2 = 0$	$c_0 = \hat{+}(c_1, c_2)$ $c_1 = \top$ $c_2 = \hat{+}(c_3, c_4)$ $c_3 = 4$ $c_4 = -4$

In the above table one can see how the term rewrite induces a rewrite on the analysis equations: the equations c_0, c_1 associated with the context $+(\mathsf{a}, [\,])$ are unchanged; the equation c_2 for the redex 0 has been altered, and new equations have been added for the subexpression $+(4, -4)$.

For each rewrite rule, the proof technique has two parts: (1) showing that the rewrite induced on the analysis equation has no effect on the solution to analysis variables associated with the context; and (2) showing that the transformation step eliminates the code added by the term rewrite.

For the arithmetic example, the transformation step $\mathcal{T}[\![\cdot]\!]$ is given by these rules:

$$\mathcal{T}[\![{}^{c_0}n]\!] = n \tag{2}$$

$$\mathcal{T}[\![{}^{c_0}\mathsf{x}]\!] = \mathsf{x} \tag{3}$$

$$\mathcal{T}[\![{}^{c_0} + ({}^{c_1}e_1, {}^{c_2}e_2)]\!] = \begin{cases} c_0 & \textit{if } c_0 \sqsubset \top;\ \textit{else} \\ \mathcal{T}[\![e_1]\!] & \textit{if } c_2 = 0;\ \textit{else} \\ \mathcal{T}[\![e_2]\!] & \textit{if } c_1 = 0;\ \textit{else} \\ +(\mathcal{T}[\![e_1]\!], \mathcal{T}[\![e_2]\!]) & \end{cases} \tag{4}$$

The rule Eqn. (4) says: if an addition expression always has a constant value, replace it by that constant; otherwise, if one of the arguments is always zero, replace the addition by the other argument.

To prove that the optimizer undoes any sequence of rewrites in a single step, we use induction over the number of rewrites. Before we begin the proof, we define some terms.

3.1 Consistency of Equations

We define a notion of *consistency* between systems of analysis equations. Intuitively, a system S_a is consistent with S_b if the system $X_b = E_b(X_b)$ has some extra variables compared to $X_a = E_a(X_a)$, but the variables they have in common map to the same program points and have the same lfp solution.

Definition 3. *Let $S_a = (D_a, E_a)$ and $S_b = (D_b, E_b)$ be two systems and $D_a = (D_1, \ldots, D_n)$. We say S_a and S_b are consistent and write $S_a \preceq S_b$ if for some permutation of applied to the elements of both D_b and E_b, $D_b = (D_1, \ldots, D_n, D_{n+1}, \ldots, D_m)$, $m \geq n$ and $x_i^\star(S_a) = x_i^\star(S_b)$ for $1 \leq i \leq n$ and the first n components of E_a and E_b map to the same program points.*

The consistency relation $\preceq$ is transitive and reflexive.

To guarantee the lfp is the same for common variables, we often need to reason about the fixpoint construction itself, i.e. the Kleene sequence $\bot, E(\bot), E(E(\bot)), \ldots$, lfp E. We define *ascending solution chains*, which generalize the Kleene sequence to arbitrary subterms of a system $X = E(X)$.

Definition 4. *Let $S = (D, E)$ be a system. The ascending solution chain of E is the set $\{X_i \in D \mid i = 0, 1, \ldots\}$ with $X_0 = \bot$, $X_{i+1} = E(X_i)$. The solution*

$X^\star = \text{lfp}\ E$ *is the greatest element of the ascending solution chain. For a function* $f : D \to D'$ *where* D' *is a complete lattice, we define* $\text{asc}\ f = \{f(X_i) \mid i = 0, 1, \ldots\}$.

We have found two techniques useful for proving consistency. The first is to make use of results that show certain transformations on systems of equations preserve the fixpoint; such transformations have been studied systematically in [23]. We state three useful results here:

Lemma 1. *Let* $S = (D, E)$ *be a system of equations. The following rewrites on* $X = E(X)$ *result in a system* $S' = (D', E')$ *satisfying* $S \preceq S'$*:*

1. *Adding an equation* $y = e_y(X)$*, where* y *is a new variable and* e_y *is any monotone function;*
2. *For an equation* $x_i = e_i(x_1, \ldots, x_n)$*, choosing some subterm* e' *of* e_i*, adding an equation* $y = e'$ *and replacing the occurrence of* e' *in* e_i *with* y*, where* y *is a new variable;*
3. *For an equation* $x_i = e_i(x_1, \ldots, x_n)$*, replacing a subterm* e' *of* e_i *with* $h(e')$*, where* h *is an identity over the lattice of* e' *for the ascending solution chain of* E*.*

The second technique for proving consistency is reason directly about the fixpoint of S'. In doing so, chaotic iteration [3,7] – the principle that fixpoints can be constructed by updating variables in any order until convergence – is useful.

3.2 Proof for the Arithmetic Example

The nullspace proof is structured as follows: (1) a lemma for each rewrite rule; (2) a theorem that any number of rewrites is undone in a single step by the optimizer. The proof structure is highly modular: one can add rewrite rules by adding a new lemma, and the rest of the proof is unaffected.

For the arithmetic example, we consider rule g_1 first. From now on, when we write $\mathcal{T}[\![\cdot]\!]$, we imply that $\mathcal{T}$ is using the lfp solution to the analysis equations.

Lemma 2. *Let* e *be an expression,* $(S, M) = \mathcal{A}[\![e]\!]$ *and* $(S', M') = \mathcal{A}[\![g_1 e]\!]$*. Then* $S \preceq S'$ *and* $\mathcal{T}[\![e]\!] = \mathcal{T}[\![g_1 e]\!]$*.*

Proof. We consider the rewrites induced on the analysis equations by application of g_1:

$$^{c_0}n \mapsto {}^{c_0}+({}^{c_1}n_1, {}^{c_2}n_2) \qquad \textit{where } n_1 + n_2 = n$$

This rewrite is applied to a subexpression inside some larger context; we write $X = E(X)$ for the unknown analysis equations associated with the rewrite context. The systems S and S' are:

S	S'
$X = E(X)$	$X = E(X)$
$c_0 = n$	$c_0 = \hat{+}(c_1, c_2)$
	$c_1 = n_1$
	$c_2 = n_2$

Consistency. In the system S, we have the equation $c_0 = n$, so clearly $c_0^\star = n$. In the system S', we have $c_1^\star = n_1$ and $c_2^\star = n_2$ and hence $c_0^\star = \hat{+}(n_1, n_2) = n_1 + n_2 = n$. Therefore $S \preceq S'$.

Transformation. To show $\mathcal{T}[\![e]\!] = \mathcal{T}[\![g_1 e]\!]$, we consider the context and redex of the rewrite separately. Since $S \preceq S'$, the analysis variables guiding transformation of the rewrite context will have the same solution in S and S'; therefore $\mathcal{T}[\![e]\!] = \mathcal{T}[\![g_1 e]\!]$ for the rewrite context. Next we consider the redex. In the original expression e we have $\mathcal{T}[\![{}^{c_0^\star} n]\!] = n$ by Eqn. (2). In the rewritten expression $g_1 e$, we have $\mathcal{T}[\![{}^{c_0^\star} + (n_1, n_2)]\!] = c_0^\star = n$ by Eqn. (4). Therefore $\mathcal{T}[\![e]\!] = \mathcal{T}[\![g_1 e]\!]$. □

Lemma 3. *Let e be an expression, $(S, M) = \mathcal{A}[\![e]\!]$ and $(S', M') = \mathcal{A}[\![g_2 e]\!]$. Then $S \preceq S'$ and $\mathcal{T}[\![e]\!] = \mathcal{T}[\![g_2 e]\!]$.*

Proof. We consider the rewrites induced on the analysis equations by application of g_2:

$$ {}^{c_a} e \mapsto {}^{c_0} + ({}^{c_a} e, {}^{c_1} 0) $$

Equations in the original system S which refer to c_a will refer to c_0 after rewriting. We use the notation $[c_0/c_a]$ to indicate this substitution:

S	S'
$X = E(X)$	$X = E(X)[c_0/c_a]$
	$c_0 = \hat{+}(c_a, c_1)$
	$c_1 = 0$

Consistency. In S' we have the equation $c_1 = 0$. Therefore $c_1^\star = 0$. From the definition of $\hat{+}$ (Eqn. (1)), we have $\hat{+}(c_a, 0) = c_a$. Using chaotic iteration, the fixpoint could be constructed by evaluating c_1 first; in subsequent iterations we could always follow an evaluation of c_a by an evaluation of c_0 to maintain the invariant $c_0 = c_a$. Therefore $c_0^\star = c_a^\star$, and $S \preceq S'$.

Transformation. Since $S \preceq S'$, $\mathcal{T}[\![e]\!] = \mathcal{T}[\![g_2 e]\!]$ for the rewrite context. For the redex, we consider cases over $c_a^\star$:

- Case $c_a^\star = \top$: then $c_0^\star = \top$ by Eqn. (1), and $\mathcal{T}[\![{}^{c_0^\star} + ({}^{c_a^\star} e, {}^{c_1^\star} 0)]\!] = \mathcal{T}[\![{}^{c_a^\star} e]\!]$ by Eqn. (4).

- Case $c_a^\star \sqsubset \top$: then $\mathcal{T}[\![{}^{c_0^\star}+({}^{c_a^\star}e, {}^{c_1^\star}0)]\!] = c_a^\star$ by Eqn. (4). Next we show $\mathcal{T}[\![{}^{c_a^\star}e]\!] = c_a^\star$ by cases over e:
  - Case $e = {}^{c_a}n$: then $c_a^\star = n$ and $\mathcal{T}[\![{}^{c_a^\star}n]\!] = n = c_a^\star$ by Eqn. (2).
  - Case $e = {}^{c_a}x$: then $c_a^\star = \top$ which contradicts the assumption $c_a^\star \sqsubset \top$, so this case does not apply;
  - Case $e = {}^{c_a} + (e_1, e_2)$: then $\mathcal{T}[\![{}^{c_a^\star} + (e_1, e_2)]\!] = c_a^\star$ by Eqn. (4).

Since $S \preceq S'$, the rewrite context will be transformed identically in both e and $g_2 e$; therefore $\mathcal{T}[\![e]\!] = \mathcal{T}[\![g_2 e]\!]$. □

We now prove that any sequence of rewrites is undone in a single step by the optimizer.

Theorem 1. *Let e be an expression, $G = \{g_1, g_2\}$ be the rewrite rules, $\mathcal{A}$ be the analysis, and $\mathcal{T}$ be the transformation. Let S be the analysis system for e. Let $g^n \cdots g^2 g^1$ be a sequence of rewrites, $g^i \in G$. Let S_n be the analysis system produced by $\mathcal{A}[\![g^n \cdots g^1 e]\!]$. Then $S \preceq S_n$ and $\mathcal{T}[\![g^n \cdots g^1 e]\!] = \mathcal{T}[\![e]\!]$.*

Proof. By induction over the number of rule applications. The base case of zero rule applications is trivial since $\preceq$ is reflexive and $\mathcal{T}[\![e]\!] = \mathcal{T}[\![e]\!]$. For the induction step, let e' be an expression and let S' be the analysis system for e'. Assume $S \preceq S'$ and $\mathcal{T}[\![e']\!] = \mathcal{T}[\![e]\!]$. We consider applying a single rewrite $g \in G$ to e'. Let S'' be the analysis system for ge'. By Lemma 2 or 3, $S' \preceq S''$ and $\mathcal{T}[\![ge']\!] = \mathcal{T}[\![e']\!]$. □

Corollary 1. *For any sequence of rewrites $g^n \cdots g^2 g^1$, $g^i \in G$ and any program p, $\mathcal{O}[\![g^n \cdots g^2 g^1 p]\!] = \mathcal{O}[\![p]\!]$.*

4 Nullspace Proofs for Imperative Languages

In the full version of this paper [18] we prove the nullspace property for a model optimizer of an imperative language. We summarize the language and the main results in this section.

4.1 An Intermediate Language

Optimizing compilers typically lower the source form to a highly constrained intermediate language (IL) over which optimizations are performed. Figure 1 shows the intermediate language we consider. This IL retains high-level control structures to simplify the analyses and proof. Expressions are syntactically constrained to be in quadruple (e.g. [9]) or A-normal [12] form; this disallows nested expressions such as $\mathsf{f}(3+4)$ in favour of $\mathsf{a} := 3+4; \mathsf{f}(\mathsf{a})$. Control structures are those typical of structured imperative languages: loops, exceptions, and if expressions.

Top-level definitions	d ::=	`function` $v_0(v_1, \ldots, v_n)\ b$	function definition
Block	b ::=	s ; $\cdots$; s; f	
Statement	s ::=	e	
		| $v := e$	variable definition
Final statement	f ::=	e	
		| `return` t	function return
		| `break`	break from loop
		| `throw` t	raise an exception
Expression	e ::=	t	trivial
		| p ($t, \ldots, t$)	primitive
		| t ($t, \ldots, t$)	function call
		| `if` t `then` b `else` b	if expression
		| `try` b `catch(`v`)` b	exception handling
		| `loop` b	loops
Trivial expression	t ::=	n	integer literal
		| v	variable use

Fig. 1: Grammar for the intermediate language.

4.2 De-optimizing Rules

We consider the following set of simple de-optimizing rules. These rules are somewhat weak, deliberately so to make the proof of reasonable size. Each rule applies only if the resulting program would pass syntactic and semantic checks; when we say "b is any block", this is shorthand for "b is any syntactically valid block containing no variables unbound in the context in which it is to be placed." We define a *subblock* of a block $e_1; e_2; \ldots ; e_n$ to be a non-empty sequence $e_i; \ldots ; e_j$ with $1 \leq i \leq j \leq n$. We write $p \mapsto p'$ to indicate a single rewrite, and $p \mapsto^* p''$ to indicate a sequence of rewrites.

- Rule g_1: $\boxed{e \mapsto \mathsf{if\ 0\ then}\ e'\ \mathsf{else}\ e}$ where e is any subblock, and e' is a new block free of abruptions.[2]
- Rule g_2: $\boxed{C[e] \mapsto \begin{array}{c}\mathsf{x} := e; \\ C[\mathsf{x}]\end{array}}$, where $C[e]$ is a statement decomposed into a context C and a subexpression e, the subexpression e is a trivial expression,[3] and x is a new globally unique identifier.
- Rule g_3: Insert at any statement position: $\boxed{\mathsf{x} := \mathsf{ref}(e)}$ where x is a new globally unique identifier and e is a trivial expression; and insert at $n \geq 0$

[2] This assumption is not necessary, but makes the proof more manageable.
[3] A trivial expression is either a variable use or a literal (Figure 1).

statement positions in the static dominance region of the definition of x: $\boxed{\mathsf{setref}(\mathsf{x}, e_i)}$ where the e_i are trivial expressions, $i = 1 \ldots n$. The primitive $\mathsf{ref}(e)$ allocates a box on the heap and stores in it the value of expression e; $\mathsf{deref}(e)$ dereferences a box, and $\mathsf{setref}(e_1, e_2)$ overwrites the value in a box.[4]

– Rule g_4: $\boxed{e \mapsto \mathsf{try}\ e\ \mathsf{catch}(\mathsf{x})\ e'}$ where e is any subblock free of function calls and throw statements, x is a new globally unique identifier, and e' is a new block free of abruptions.

By applying these rules to a program that computes $+(1, 2)$ we can obtain an elaborate version that has a debugging mode, boxes the integers, and handles exceptions:[5]

```
function main()        ↦*   function main()
   r := +(1, 2)                debugMode := 0
   return r                    a := 1
                               b := 2
                               x := ref(a)
                               y := ref(b)
                               if debugMode then
                                 print(a)
                                 print(b)
                               else
                                 try
                                   r := +(a, b)
                                   return r
                                 catch(err)
                                   g := deref(x)
                                   h := deref(y)
                                   print(g)
                                   print(h)
                                   exit(1)
```

Our optimizer is guaranteed to undo all the rule applications, effectively reversing the direction of the arrow $\mapsto^*$

An obvious question is: why pose the rewrites as de-optimizing rules? Why not reverse the direction of the rewrites and apply the inverse rules directly to

[4] This is a weak form of a "box any variable" rule, weakened to avoid dealing with propagation through boxes. Propagation through boxes is a complex analysis beyond the scope of this paper. Instead we allow any value to escape into a box, and guarantee that if the box is never read, then the boxing will be undone in the transformation step, and other optimizations will be performed as if the value did not escape.

[5] To avoid propagation through boxes, we have manually applied this optimization, replacing uses of c and d with a and b, respectively. Also, in this example we assume print and exit are primitives.

programs? The problem is this: deciding whether some of the inverse rules apply requires a supporting analysis. For example, deciding whether Rule g_3^{-1} applies to a given box requires knowing that any reads from the box are unreachable and that the box does not escape. Doing such analyses for every rewriting step would be inefficient; and devising a rewriting strategy that would allow a convergence proof for a general compiler – with interprocedural analysis and heap analysis – appears difficult. Instead, we prove that our optimizer effectively subsumes a rewrite system implementing the inverse rules. If the the optimizer runs in $O(|p| \log |p|)$ time, then the rewrites are undone in $O(|p| \log |p|)$ time.

4.3 Analysis Equations

We limit our attention to "simplifying" optimizations that reduce or eliminate code. Optimizations that reorder code – for example, loop nest optimizations – are interesting but we regard them as an orthogonal issue.

We formulate the analyses as an optimistic superanalysis [22,1]. An optimistic analysis is one that starts with the optimistic assumption of $\perp$ and iterates to a least fixpoint. In an optimistic analysis, intermediate results during a fixpoint iteration are not conservative and may not be used for transformation. We adopt the terminology of [8] in calling simultaneous, interacting analyses a "superanalysis." We consider four simultaneous analyses:

- *Reachability* decides whether expressions may be reached during execution. Unreachable expressions are dead code and may be eliminated.
- *Congruence* partitions program points into equivalence classes based on whether the values reaching them are always the same.
- *Escape*, a backward analysis that determines how a value produced at a program point may be used.
- *Transform*, a simple analysis deciding whether subexpressions are to be processed for value or effect in the transformation step.

Our analyses are for the most part conventional and well-understood analyses drawn from the literature; we make some adjustments to enable the proof.

The proof follows the pattern developed in Section 3: a lemma for each rewrite rule, followed by a theorem proven inductively over the number of rewrites. The use of superanalysis – that is, simultaneous analyses – appears essential to proving consistency of systems of equations after a rewrite. We refer the reader to an expanded version of this paper for details [18].

5 Nullspace Compilers Are True Optimizers

In this section, we show a consequence of a nullspace proof: that under some weak assumptions, optimized programs are "minimal" in some metric.

We review some necessary rewrite terminology [4,5]. We write $x \mapsto y$ to mean that x rewrites to y in a single step, and $\overset{*}{\mapsto}$ ("derives") for the reflexive, transitive closure of $\mapsto$ We write $x \leftrightarrow y$ ("convertible") to mean $x \mapsto y$ or $y \mapsto x$, and $\overset{*}{\leftrightarrow}$

for the reflexive, transitive closure of $\leftrightarrow$. Then $\overset{*}{\leftrightarrow}$ is an equivalence relation and partitions the set of programs into equivalence classes. We use the notation $[p]$ for the equivalence class of a program p under $\overset{*}{\leftrightarrow}$.

Remark. The nullspace theorem guarantees $\mathcal{O}[\![p]\!] = \mathcal{O}[\![p']\!]$ for all $p' \in [p]$.

We write $|p|$ for the textual size of a program. We make three assumptions:

1. Rewrites strictly increase size: $|p| < |gp|$ for a rewrite g and program p.
2. $\mathcal{O}$ is nonincreasing in program size: $|\mathcal{O}[\![p]\!]| \leq |p|$.
3. For any p, $\mathcal{O}[\![p]\!]$ is a fixpoint: $\mathcal{O}[\![\mathcal{O}[\![p]\!]]\!] = \mathcal{O}[\![p]\!]$.

The third assumption is reasonable for a transformation $\mathcal{T}[\![\cdot]\!]$ that makes the minimal changes to support the nullspace proof. However, we allow the possibility that more improvements are performed by $\mathcal{T}[\![\cdot]\!]$ than required by the nullspace proof.

Lemma 4. *Let p be a program. There is a nonempty set of "least programs" $P_0 \subseteq [p]$ such that (1) for every least program $p_0 \in P_0$ there is no $z \in [p]$ such that $z \mapsto p_0$; (2) p is derivable from some $p_0 \in P_0$.*

Proof. Since $|p| < |gp|$ for any rewrite g and program p, the relation $\overset{*}{\mapsto}$ is antisymmetric (i.e. there are no cycles $p_1 \mapsto p_2 \mapsto \ldots \mapsto p_n \mapsto p_1$). Furthermore, $\overset{*}{\mapsto}$ is reflexive and transitive by definition. Therefore $([p], \overset{*}{\mapsto})$ is a poset, and the least programs $P_0 \subseteq [p]$ are simply the least elements of the poset. □

Definition 5. *The* abstraction level *of a program p, written $AL(p)$, is the minimum over all least programs $p_0 \in [p]$ of the number of rewrite steps in the shortest derivation $p_0 \mapsto \ldots \mapsto p$.*

Theorem 2. *For all programs p, $AL(\mathcal{O}[\![p]\!]) = 0$.*

Proof. (By contradiction). Choose p, and assume $AL(\mathcal{O}[\![p]\!]) > 0$. Then there is a program p' and rewrite g such that $gp' = \mathcal{O}[\![p]\!]$. Apply the optimizer again: $\mathcal{O}[\![gp']\!] = \mathcal{O}[\![p']\!] = \mathcal{O}[\![\mathcal{O}[\![p]\!]]\!] = \mathcal{O}[\![p]\!]$ by the nullspace theorem and the fixpoint assumption. But then $|\mathcal{O}[\![p]\!]| = |\mathcal{O}[\![p']\!]| \leq |p'| < |gp'| = |\mathcal{O}[\![p]\!]|$, which is a contradiction. □

Thus, an optimizing compiler with the nullspace property and satisfying the above assumptions is a *true optimizer*: it produces a program which is minimal in the metric AL.

6 Conclusion

We have demonstrated a technique for proving guaranteed optimization for imperative compilers. The "de-optimizing" rules used by our proof technique give programmers an intuitive feel for what abstractions they may introduce without performance loss. The proof shows that any sequence of de-optimizing rule applications is undone by a single application of the optimizer; hence, the optimizer effectively subsumes a rewrite system implementing the inverse (optimizing) rules. The proof relies on adopting a superanalysis approach to optimization – that is, simultaneous analyses followed by a single transformation step.

A key unanswered question is: will the approach extend to more complex analyses and transformations? We believe so, within limits. We identify four requirements for adding to the framework: (1) the analysis must be posable as equations in a lattice framework; (2) the analysis equations must be composable – that is, if one assembles an expression from subexpressions, there is a corresponding way to assemble the analyses of subexpressions; (3) one must be able to describe suitable de-optimizing rewrite rules; and (4) the rewrite rules must admit a proof along the lines of Lemmas (2,3). We have explored – although not yet in a rigorous way – adding function pointers, interprocedural analysis and propagation of values through nonmutable heap objects, and these appear to meet the requirements. In future work we plan to incorporate them formally into our proof framework.

Acknowledgments. We thank Kenneth Chiuk, Kent Dybvig, Ron Garcia, Jaakko Järvi, Amr Sabry, Jeremy Siek and Jeremiah Willcock for their comments on drafts of this paper, Arch Robison for discussions, and the reviewers for helpful comments.

References

1. Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.
2. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Principles of Programming Languages*, pages 238–252, January 1977.
3. Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 1–12, 1977.
4. Nachum Dershowitz. A taste of rewrite systems. In *Proc. Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 199–228, 1993.
5. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. 1990.

6. A. J. Gill and S. L. Peyton Jones. Cheap deforestation in practice: An optimiser for Haskell. In Bjørn Pehrson and Imre Simon, editors, *Proceedings of the IFIP 13th World Computer Congress. Volume 1 : Technology and Foundations*, pages 581–586, Amsterdam, The Netherlands, August 28–September 1 1994. Elsevier Science Publishers.
7. Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206. ACM SIGACT and SIGPLAN, ACM Press, 1973.
8. Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. *ACM SIGPLAN Notices*, 31(1):270–282, January 2002.
9. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 2000.
10. Matthias Müller. Abstraction benchmarks and performance of C++ applications. In *Proceedings of the Fourth International Conference on Supercomputing in Nuclear Applications*, 2000.
11. Arch D. Robison. The abstraction penalty for small objects in C++. In *POOMA'96: The Parallel Object-Oriented Methods and Applications Conference*, Feb. 28 - Mar. 1 1996. Santa Fe, New Mexico.
12. Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3–4):289–360, 1993.
13. David Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.
14. Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A unifying framework for numerical linear algebra. In *Parallel Object Oriented Scientific Computing*. ECOOP, 1998.
15. Alex Stepanov. Abstraction penalty benchmark, 1994.
16. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. *ACM SIGPLAN Notices*, 32(12):203–217, 1997.
17. Todd L. Veldhuizen. Arrays in Blitz++. In *Computing in Object-Oriented Parallel Environments: Second International Symposium (ISCOPE 98)*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230. Springer-Verlag, 1998.
18. Todd L. Veldhuizen and Andrew Lumsdaine. Guaranteed optimization: Proving nullspace properties of compilers. Technical Report TR564, Indiana University Computer Science, 2002.
19. Eelco Visser. A survey of rewriting strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.
20. Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).
21. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.
22. Mark N. Wegman and Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.
23. Jiawang Wei. Correctness of fixpoint transformations. *Theoretical Computer Science*, 129(1):123–142, June 1994.

Representation Analysis for Coercion Placement

Karl-Filip Faxén

Dept. of Microelectronics and Information Technology, Royal Institute of Technology,
Electrum 229, S-164 40 Kista, tel: +46 8 790 41 20
kff@it.kth.se

Abstract. This paper presents a global approach to representation analysis based on program-wide data and control flow information. Boxing and unboxing coercions can be placed around any variable occurrence, not only where values are produced and consumed.
The analysis first constructs a graph representing all legal coercion placements, then selects one of them. Assigning unboxed representations to as many variables as possible does not necessarily minimize execution time (or the number of executed coercions); we present and measure several heuristics.
When combined with function cloning, the analysis is powerful enough to eliminate almost all coercions from several nonstrict programs, including a simple polymorphic type checker.

1 Introduction

In an untyped language like Scheme, or a polymorphically typed language like SML or Haskell, the compiler must be able to generate code to manipulate a value without knowing what kind of value it is; integer, cons cell, function closure etc. For example, when compiling the function `fst`, which extracts the first component of a pair, the compiler has to know the size of the value (what kind of load instruction to use when fetching it) and how it should be returned to the caller (e.g. in an integer or floating point register). In a nonstrict language such as Haskell, an additional complication is that even if it is known that a value is of type `Int`, it may be unevaluated and represented by (a pointer to) a delayed computation called a *thunk*.

Many compilers deal with these restrictions by keeping all values in memory and representing them by pointers. Since this uniform *boxed* representation incurs a significant overhead for simple data like (evaluated) integers and since most programs contain parts, such as intermediate results in arithmetic expressions and strict function arguments and results, where boxing is not necessary, it is possible to use more efficient *unboxed* representations for some values. For each unboxed representation σ (*e.g.* `int` or `float`) there are two *coercion operators* `box_`σ and `unbox_`σ which convert values between σ and `box`.

This raises two interesting questions, which are the subject of this paper: For which variables *can* an unboxed representation be used? And for which of those variables *should* an unboxed representation be used? Many papers have been devoted to the first question [12,17,10,11,8], but much fewer have addressed

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 278–293, 2002.
© Springer-Verlag Berlin Heidelberg 2002

the second one (Peterson addresses the problem in [15] but does not discuss interprocedural issues and Henglein&Jørgensen touch on it in [11]).

Most often, however, that question is implicitly answered by "all variables that the present method finds legal to unbox". If coercions can only be placed where values are produced (constants, operator applications) or consumed (operator arguments), this is the best that can be done. Conceptually, these techniques [3,10] start with a canonically boxed program, where every value is boxed immediately after being produced and unboxed immediately before being consumed, and then remove as many coercions as possible.

```
(1a) let rept x n = if n>0 then Cons x (rept x (n-1)) else Nil
     in rept (box_int z) 1000
(1b) let rept x n = if n>0 then Cons (box_int x) (rept x (n-1))
                                else Nil
     in rept z 1000

(2a) let foo x y = if y>0 then foo (id x) (y-1) else unbox_int x + y
     in foo z 10
(2b) let foo x y = if y>0 then foo (unbox_int (id (box_int x))) (y-1)
                               else x + y
     in foo (unbox_int z) 10
```

Fig. 1. Examples of box-unbox cycles and loss of box sharing

If, on the other hand, coercions can be placed anywhere, there are situations where it might be better to let a variable be boxed even if it might legally be unboxed. For instance, an unboxed value might be boxed several times: Suppose the elements of the resulting list in Fig. 1, upper half, must be boxed. Then alternative (1a) is better than alternative (1b) even though `x` is unboxed in the latter version. A value may also be repeatedly boxed and unboxed: Suppose that the argument (and return value) of `id` must be boxed in the lower half of Fig. 1. Then (2a) is better than (2b) even though `x` is unboxed in the latter version.

In this paper we conceptually start from a program with coercions around every subexpression and we then choose representations for variables and subexpressions so that as many coercions as possible become redundant. We represent the set of legal representation assignments as a graph where the nodes are sets of variables and subexpressions which must have the same representation and the edges correspond to sets of coercions.

The rest of the paper is organized as follows: Section 2 presents the analyzed language Fleet, section 3 presents the coercion placement technique in more detail, section 4 presents some experimental results, section 5 discusses related work and section 6 concludes and outlines future work.

```
letrec fac = λ^i m.λ^ii n. let c = positive^iii n^1
                           in if c^2 then let mn = m^3 *^iv n^4
                                          dn = dec^v n^5
                                          g = fac^6 mn^7
                                          r = g^8 dn^9
                                       in r^10
                                  else m^11

in let x = thunk^vi let ten = 10^vii
                    in ten^12
       t = thunk^viii let one = 1^ix
                          ex = eval x^13
                          h = fac^14 one^15
                          tr = h^16 ex^17
                      in tr^18
   in t^19
```

`positive` x returns `1` (true) if $x > 0$.
`dec` x returns $x - 1$.

Fig. 2. Tail recursive factorial in Fleet

2 Fleet

Figure 4 presents Fleet. It is a simple functional language containing the constructs of the lambda calculus as well as constructors, case expressions, built-in operators and recursive and nonrecursive let. Graph reduction is handled by explicit `eval` and `thunk` constructs; the rest of the language has a call-by-value semantics. This makes Fleet suitable as an intermediate language for strict as well as nonstrict languages, and in the latter case, the effect of strictness analysis or directives is readily expressed.
The labels which occur in the syntax play two different roles:

- They identify *producers* of values; these are lambda abstractions, operator applications, constructor applications, and thunks. Producers are precisely the constructs which create new nodes (values) in the operational semantics in Fig. 4. Nodes carry the label of their producers. We will use lower case Roman numerals for producer labels ($i, ii, iii, \ldots$).
- They identify *variable occurrences*; this is where coercions will be inserted. The use of labels makes it possible to distinguish between different occurrences of the same variable. We will use Arabic numerals for occurrence labels ($1, 2, 3, \ldots$).

Figure 2 shows a rather contrived example program in Fleet including a tail recursive factorial function (we have taken a few syntactic liberties, namely `let` expressions with several bindings and `if` as syntactic sugar for a boolean case).

`fac` : $\{i\}$	`m` : $\{iv, ix\}$	`n` : $\{v, vii\}$	`c` : $\{iii\}$	`mn` : $\{iv\}$
`dn` : $\{v\}$	`g` : $\{ii\}$	`r` : $\{iv, ix\}$	`t` : $\{iv, viii, ix\}$	`one` : $\{ix\}$
`ex` : $\{vii\}$	`h` : $\{ii\}$	`tr` : $\{iv, ix\}$	`x` : $\{vi, vii\}$	`ten` : $\{vii\}$

Shared variables: {`fac`, `n`}

Fig. 3. Flow information and shared variables for factorial

2.1 Flow and Sharing Information

The analysis presented in this paper relies on the result of a flow analysis performed prior to the representation analysis. The method used to compute the flow information is not important as long as it satisfies the correctness criterion in Section 2.2. Flow information tells us something about the values a variable can be bound to (or a subexpression evaluate to). In our case, it tells us where in the program the values were produced, with the producers identified by their labels. As any result of static analysis, flow information is approximate, but it is *safe* as long as it gives a superset of the actual values.

Figure 3 gives safe flow information for the program in Figure 2. For instance, it indicates that the value of the variable `m` is either the constant $\mathtt{1}^{ix}$ or the result of the multiplication operator $\mathtt{*}^{iv}$. Higher order control flow information is also captured. Although this is a first order program, making the control flow rather trivial, one can see that the flow information for the variable `g` is $\{ii\}$ which indicates that it will always be bound to the λ`n` abstraction. Note that the flow information for the variable `t` includes both the label of the `thunk` construct ($viii$) and the labels corresponding to the body of the thunk (iv, ix). This reflects the fact that the thunk will be overwritten with the result of evaluating it.

Formally, flow information takes the form of a function φ, called a *flow assignment*, from the variables in the program to sets of (producer) labels.

Sharing information is also used as a guide when deciding where to place coercions. The information takes the form of a set of variables which may be bound to shared objects at run-time.

An object is shared when it is simultaneously stored in multiple environments or under different names in the same environment. Shared objects may be accessed more than once, unshared objects are only accessed once.

There are several ways to compute flow and sharing information. The method we use is presented in [3]. Briefly, the flow analysis is based on an annotated polymorphic subtype inference system, and the sharing information is computed by post processing the flow information. Many other flow analyses exist; see for instance Mossin [14] for an overview of several type based alternatives.

2.2 The Semantics of Fleet

We give an operational semantics for Fleet in Fig. 4. The semantics is in the style of natural semantics and depends on a flow assignment φ.

$$
\begin{array}{lll}
e \in \text{Expr} & \to x \mid x_1\, x_2 \mid \lambda^l v.\, e \mid C^l\, x_1 \ldots x_r \mid op^l\, x_1 \ldots x_r \\
& \mid \texttt{case}\ x\ \texttt{of}\ alt_1; \ldots;\ alt_n\ \texttt{end} \\
& \mid \texttt{let}\ v = e\ \texttt{in}\ e' \mid \texttt{letrec}\ v_1 = b_1; \ldots; v_n = b_n\ \texttt{in}\ e \\
& \mid \texttt{thunk}^l\ e \mid \texttt{eval}\ x \\
b \in \text{Build} & \to \lambda^l v.\, e \mid C^l\, x_1 \ldots x_k \mid \texttt{thunk}^l\ e \\
alt \in \text{Alt} & \to C\, v_1 \ldots v_k\ \texttt{->}\ e \\
x \in \text{VarOcc} & \to v^l \\
v \in \text{Var}, & \quad l \in \text{Label}, \qquad C \in \text{DataCon} \cup \text{IntLit} \cup \text{FloatLit}
\end{array}
$$

$$\rho \vdash G, x \to G, \rho(x) \qquad \text{VAR}$$

$$\frac{a \text{ fresh}}{\rho \vdash G, b \to G[a \mapsto (\rho, b)],\ a} \qquad \text{BUILD}$$

$$\frac{G(\rho(x_1)) = (\rho', \lambda^l v.\, e) \qquad \rho'[v \mapsto \rho(x_2)] \vdash G, e \to G', a}{\rho \vdash G, x_1\, x_2 \to G', a} \qquad \text{APP}$$

$$\frac{[\![op]\!]\, G(\rho(x_1)) \ldots G(\rho(x_r)) = C \qquad a \text{ fresh}}{\rho \vdash G, op^l\, x_1 \ldots x_r \to G[a \mapsto ([\,], C^l)],\ a} \qquad \text{OP}$$

$$\frac{G(\rho(x)) = (\rho', C^l\, x_1 \ldots x_r) \qquad \rho[v_1 \mapsto \rho'(x_1), \ldots, v_r \mapsto \rho'(x_r)] \vdash G, e \to G', a}{\rho \vdash G, \texttt{case}\ x\ \texttt{of}\ \ldots C\, v_1 \ldots v_r\ \texttt{->}\ e; \ldots\ \texttt{end} \to G', a} \qquad \text{CASE}$$

$$\frac{\rho \vdash G, e \to G', a \qquad \rho[x \mapsto a] \vdash G', e' \to G'', a'}{\rho \vdash G, \texttt{let}\ v = e\ \texttt{in}\ e' \to G'', a'} \qquad \text{LET}$$

$$\text{LETREC}$$

$$\frac{a_1 \ldots a_n \text{ fresh} \quad \rho' = \rho[\ldots, v_i \mapsto a_i, \ldots] \quad \rho' \vdash G[\ldots, a_i \mapsto (\rho', b_i), \ldots], e \to G', a}{\rho \vdash G, \texttt{letrec}\ v_1 = b_1; \ldots; v_n = b_n\ \texttt{in}\ e \to G', a}$$

$$\frac{G(\rho(x)) = (\rho', \texttt{thunk}^l\, e) \qquad \rho' \vdash G[\rho(x) \mapsto \texttt{black\_hole}^l], e \to G', a}{\rho \vdash G, \texttt{eval}\, x \to G'[\rho(x) \mapsto \texttt{ind}^l\, a], a} \qquad \text{EVAL-I}$$

$$\frac{G(\rho(x)) = \texttt{ind}^l\, a}{\rho \vdash G, \texttt{eval}\, x \to G, a} \qquad \frac{G(\rho(x)) = (\rho', b) \qquad b \text{ is not a thunk}}{\rho \vdash G, \texttt{eval}\, x \to G, \rho(x)} \qquad \text{EVAL-II,III}$$

$$\frac{\exists x \in Dom(\rho)\,.\, \mathcal{L}(G(\rho(x))) \notin \varphi(x) \text{ or } \rho(x) = wrong}{\rho \vdash G, e \to [\,], wrong} \qquad \text{WRONG}$$

Fig. 4. The syntax and semantics of Fleet

The rules in Fig. 4 allow us to prove judgments of the form $\rho \vdash G, e \rightarrow G', a$ where ρ is an environment mapping variables to addresses, G and G' are graphs mapping addresses to *nodes*, which are either *closures* (which are pairs of environments and expressions), *black holes* or *indirections*, and a is an address.

A black hole is used to mark a thunk during the evaluation of its body; an attempt to `eval` a black hole aborts the program since it is a symptom of infinite recursion. When evaluation of a thunk body is complete, the thunk is overwritten with (an indirection to) the result. Subsequent `eval`s on that thunk simply dereference the indirection.

Every node is labeled with the label of the producer of the node. When a thunk is rewritten to a black hole and then to an indirection, it retains the same label.

There is one special address, *wrong*, which signals that the flow assignment φ has been violated. The WRONG rule in the semantics is applicable whenever the environment and the graph together maps a variable v to a node with a label that is not in $\varphi(v)$ (that is, when $\mathcal{L}(G(\rho(v))) \not\in \varphi(v)$, where $\mathcal{L}(\cdot)$ maps a node to its label). It also propagates *wrong* out of the right hand sides in nonrecursive let bindings (which is the only sequencing construct in Fleet). We say that flow assignment φ is correct with respect to the program P iff $[\,] \vdash [\,], P \rightarrow [\,], wrong$ is not derivable ($[\,]$ is the empty environment or graph, respectively).

3 Representation Analysis

The representation analysis described in this paper takes a program (closed Fleet expression) P and a flow assignment φ (and optionally sharing information) and produces a representation assignment Σ which maps every variable and occurrence to a representation.

The representation assigned to a variable is the representation of the value the variable is initially bound to. Thus the representation of a formal parameter is the representation the function expects the corresponding argument to have. Similarly, the representations of the variables bound in a `case` alternative are the representations that the code unpacking the matching constructor assumes its components to have. The representation of a variable occurrence is the representation that the value of the variable will be coerced to at that occurrence (*e.g.* before being passed to a function if the occurrence is the argument part of a function application).

The representation assignment can be used to annotate variable bindings with representations and place coercions around occurrences v^l where $\Sigma(v) \not\trianglerighteq \Sigma(v^l)$ (this is where the representation of the variable differs from the representation it is used with).

The analysis is divided into two phases; first finding the set of all legal representation assignments, then selecting one of them according to a placement strategy. This division separates the problem into mechanism (how do we know which representation assignments are legal) and policy (among the legal representation assignments, which one is best).

$O(v^l) = \{v \rightsquigarrow v^l\}$
$E(\eta,\, x) = O(x) \cup \{x \to \eta\}$
$E(\eta,\, x_1\, x_2) = O(x_1) \cup O(x_2) \cup \{\langle l \rangle \to \eta, x_2 \to v \mid \lambda^l v.\, e \in \mathcal{P}(P, \varphi(x_1))\}$
$E(\eta,\, \lambda^l v.\, e) = \{l \to \eta\} \cup E(\langle l \rangle, e)$
$E(\eta,\, C^l\, x_1 \ldots x_r) = O(x_1) \cup \ldots \cup O(x_r) \cup \{l \to \eta\}$
$E(\eta,\, op^l\, x_1 \ldots x_r) = O(x_1) \cup \ldots \cup O(x_r) \cup \{l \to \eta\}$
$E(\eta,\, \texttt{case}\ x\ \texttt{of}\ alt_1; \ldots;\ alt_n\ \texttt{end}) = O(x) \cup E_A(\eta, x, alt_1) \cup \ldots \cup E_A(\eta, x, alt_n)$
$E(\eta,\, \texttt{let}\ v = e\ \texttt{in}\ e') = E(v, e) \cup E(\eta, e')$
$E(\eta,\, \texttt{letrec}\ v_1 = b_1; \ldots; v_n = b_n\ \texttt{in}\ e) = E(v_1, b_1) \cup \ldots \cup E(v_n, b_n) \cup E(\eta, e)$
$E(\eta,\, \texttt{thunk}^l\, e) = E(\eta, e)$
$E(\eta,\, \texttt{eval}\ x) = O(x) \cup \{x \to \eta\}$
$E_A(\eta,\, x,\, C\, v_1 \ldots v_r\ \texttt{->}\ e) = E(\eta, e) \cup \{x_1 \to v_1, \ldots, x_r \to v_r \mid C^l\, x_1 \ldots x_r \in \mathcal{P}(P, \varphi(x))\}$

Node equivalence for the c-graph: Let $\equiv_0$ be given by $\eta \equiv_0 \eta'$ iff there is a copy edge $\eta \to \eta'$ in the cc-graph of P and let $\equiv$ be the symmetric, reflexive and transitive closure of $\equiv_0$.
The nodes of the c-graph of P are the equivalence classes of $\equiv$ and for every coercion edge $v \rightsquigarrow x$ in the cc-graph of P there is an edge $[v] \rightsquigarrow [x]$ in the c-graph of P.

Fig. 5. The construction of the c and cc-graphs of the program P using the flow assignment φ

3.1 Finding Legal Representations

The first phase of the analysis uses flow information to find which choices of representations are legal. It is a formalization of the following rules:

1. The code which extracts the components of a heap cell must agree with the code that built the cell about the representations of the components.
2. The code which calls a function must agree with the code of that function about the representations of the formal parameters and the return value.
3. If a variable may be bound to a thunk, function closure or constructed value (pair, cons cell, ...) the representation of the variable has to be `box`. If a variable is only bound to (evaluated) integers, its representation may be `int` or `box`. If a variable is only bound to (evaluated) floating point numbers, its representation may be `float` or `box`.

The first phase represents the set of all legal assignments as a graph called the *coercion graph* (or simply the *c-graph*). The c-graph is computed in two steps.[1] First the *copy-coercion graph* (or *cc-graph*) is built, and then the c-graph is constructed by collapsing some of the edges in the cc-graph.

[1] This is for presentational reasons; it need not be implemented that way.

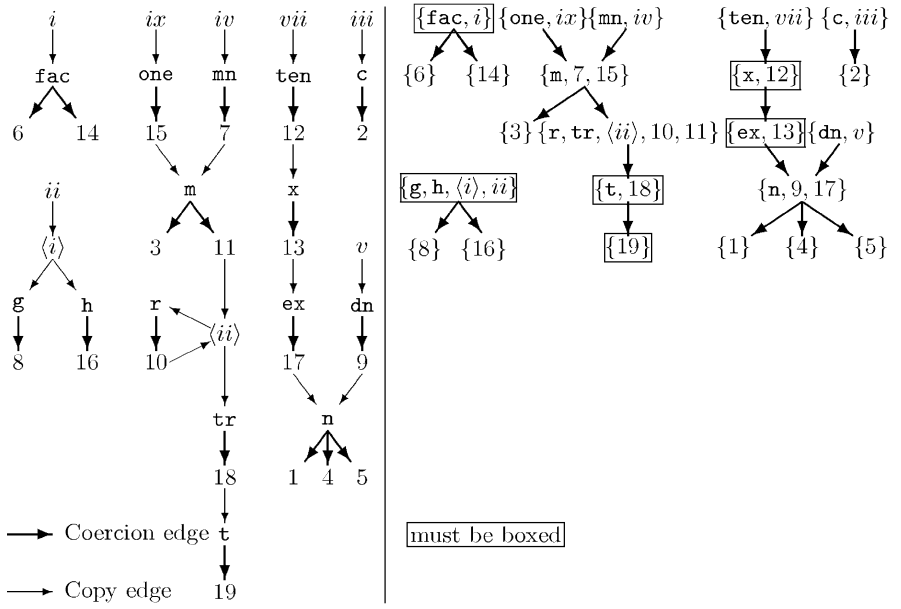


Fig. 6. The cc-graph (left) and the c-graph (right) of the factorial program (only the labels are given for occurrence nodes, *e.g.* 6 rather than $\texttt{fac}^6$). Note that coercion edges are marked as slightly heavier rather than squiggly.

The cc-graph. Figure 5 gives a formal definition of the cc-graph. A node η is either a variable, a variable occurrence, a pseudo variable $\langle l \rangle$ representing the return value of the lambda abstraction labelled with l, or a producer label.

For each variable occurrence v^l there is a *coercion edge* $v \rightsquigarrow v^l$ representing the coercion which may eventually be wrapped around this occurrence. The edges of the c-graph will be derived from the coercion edges of the cc-graph.

In the other direction we have *copy edges* going from variable occurrences, pseudo variables and producer labels to (pseudo) variables. There is an edge $v_0^l \to v$ if a value may be copied from the variable occurrence v_0^l to the variable v. Similarly, there is an edge $l \to v$ if a value may be copied from a constructor or operator application or lambda abstraction to the variable v. In either case, the representation of v and v_0^l (or l) has to be the same.

There are copy edges connecting the arguments of applications to the formal parameters of the applied functions, from the bodies of lambda abstractions to the applications where the abstraction may be applied, and from constructor applications to case alternatives. Flow information is essential for creating the copy edges in the cc-graph.

In Fig. 5, $E(\eta, e)$ gives the edges generated for e including the copy edge(s) from the result(s) of e to η while $O(x)$ denotes the coercion edge associated

with the variable occurrence x. We also write $\mathcal{P}(P, L)$ for the set of producers (abstractions, ...) in the program P labelled with some $l \in L$.

The cc-graph nodes corresponding to producer labels are called *source nodes* and the nodes corresponding to variable occurrences that form the scrutinized part of case expressions or arguments to operator applications are called *sink nodes*. They are special in that they are the only nodes for which the representation `box` may be illegal. This is the case if the operator or case expression is an unboxed one. In Fig. 6 (left) the nodes i, ii, iii, iv, v, vii and ix are all source nodes and the nodes $1, 2, 3, 4, 5, 6, 8, 14, 16$ and 19 are all sink nodes.

The c-graph. The c-graph contains the following information:

- Which variables and occurrences that have to have the same representation; two variables or occurrences are assigned to the same c-graph node iff they must have the same representation.
- For each c-graph node, which representations are legal, depending on what kind of values the variables in that node may be bound to.
- The edges of the c-graph capture the data flow between the nodes. This information is used by the heuristics in the second phase. Sharing information may optionally be associated with the c-graph.

The c-graph is constructed from the cc-graph by collapsing the copy edges. The formal definition is given in Fig. 5, and the c-graph of the factorial program is given in Fig. 6 (right).

In order to be correct, the representation assigned to a variable or a variable occurrence must be compatible with all values that can be bound to that variable or occurrence. Here we again use flow information. For a source or sink node, the only legal representation is the native representation of the type corresponding to the node (result or argument type of operator application etc). Otherwise, the representation σ is legal for $[\eta]$ if σ=`box` or if σ is the native representation of every producer $b \in \mathcal{P}(P, \varphi(\eta'))$, $\eta' \in [\eta]$. The native representation of a literal or operator application is the unboxed representation of the (return) value whereas for `thunk`s, abstractions and data constructors the native representation is `box`.

3.2 Selecting Representations

We will now look at a number of strategies for selecting representations for the unconstrained nodes of the c-graph (those nodes which have more than one legal representation). Note that the choices of representations of different unconstrained c-graph nodes are independent from the point of view of correctness. All of the strategies share one important constraint: No coercions are placed between a tail call and a return since that would turn the tail call into a non-tail call with a potentially unbounded increase in stack space usage. We implement this by collapsing c-graph edges corresponding to such coercions. We will list the strategies in order of decreasing willingness to assign unboxed representations; each strategy implies all of the constraints of the strategies listed before it.

```
letrec fac^box
  = λ^i m^box.λ^ii n^box.
    let c^int = positive^iii
                  (unbox_int n)
    in if c
       then
       let mn^int = unbox_int m
                    *^iv unbox_int n
           dn^int = dec^v (unbox_int n)
           g^box = fac (box_int mn)
           r^int = g (box_int dn)
       in r
       else m

in let x^box
     = thunk^vi let ten^int = 10^vii
                in box_int ten
     t^box
       = thunk^viii
         let one^int = 1^ix
             ex^box = eval x
             h^box = fac (box_int one)
             tr^box = h ex
         in tr
   in t
```

```
letrec fac^box
       = λ^i m^int.λ^ii n^int.
         let c^int = positive^iii n
         in if c
            then
            let mn^int = m *^iv n
                dn^int = dec^v n
                g^box = fac mn
                r^int = g dn
            in r
            else m

in let x^box
     = thunk^vi let ten^int = 10^vii
                in box_int ten
     t^box
     = thunk^viii
       let one^int = 1^ix
           ex^box = eval x
           h^box = fac one
           tr^int
           = h (unbox_int ex)
       in box_int tr
   in t
```

Fig. 7. Tail recursive factorial with coercions placed according to endpoints (left) or any one of the more aggressive strategies (right)

Any. All unconstrained nodes of the c-graph are assigned unboxed representations.

No cycles. All cycles in the c-graph are collapsed, forming the *acyclic c-graph* (or *ac-graph*). A node in the ac-graph is unconstrained if all c-graph nodes that node corresponds to are unconstrained; all unconstrained ac-graph nodes are assigned unboxed representations.

The rationale for this strategy is as follows: Cycles in the c-graph imply that a value is copied between the nodes an unlimited number of times (c-graph cycles are created by recursion in the analysed program). If the nodes are assigned different representations, the same value may be boxed and unboxed an unbounded number of times; it would be better to keep it boxed all the time in this case.

No rebox. The same value may not be boxed, then unboxed, then boxed again. Representations are assigned in such a way that there is no path from a boxed node, over a number of unboxed nodes, to a boxed node. This is a stronger version of the previous strategy, which only disallowed paths with an unbounded number of coercions.

Box sharing. In addition to the no rebox restriction, a boxed node is not allowed to have an unboxed predecessor if any of the variables or occurrences in the boxed node can be bound to shared values. This condition will in general force boxing to happen earlier, e.g. before a call to a recursive function. It guarantees that, at any time, there will be no more reachable boxes in the heap than it would have been if no boxing optimizations had been performed.

Figure 7 (left) gives the factorial program as optimized by the box sharing strategy. The more aggressive strategies give the same result for this program.

Endpoints. Coercions are only placed on source and sink edges in the c-graph. Thus, every edge that is not a source or sink edge is collapsed.

This has the consequence that unboxing coercions only occur when the unboxed value will be immediately consumed (e.g. in the argument position of an operator application) and boxing coercions only occur when a value has just been created (e.g. by an operator application). This rule guarantees that there will be no more coercions executed than if the program had not been optimized.[2]

Figure 7 (right) gives the factorial program as optimized by the endpoints strategy. This program can of course be improved by common subexpression elimination within each function.

4 Implementation

We have implemented coercion placement in an experimental compiler for a simple lazy higher order functional language called Plain. The compiler has a (rather primitive) strictness analyzer based on demand propagation and a (rather sophisticated) flow analyzer based on polymorphic subtype inference [3]. Except being used for coercion placement, the flow information is exploited by update avoidance and cheap eagerness optimizations (the latter is an instance of speculative evaluation; if an expression is cheap to evaluate and guaranteed not to yield a run-time error, there is no need to build a thunk for it [5]). Since several optimizations are program-wide, separate compilation is not supported. The compiler is described more fully in the author's PhD thesis [4].

The compiler does not support floating point numbers; the only representations are `box` and `int`.

In contrast to e.g. [12,17], unboxed objects can be stored in data structures, even in fields with polymorphic type. This is implemented as follows: A heap cell

[2] At least if local common subexpression elimination is applied and no variable let-bound to a producer is free in a lambda abstraction.

consists of a header word, which is a pointer to a statically allocated descriptor containing the information used by the garbage collector, followed by pointer and nonpointer data. There may be several descriptors for a single constructor; heap nodes built with the same constructor but different representations of the components have different headers and descriptors.

Case expressions still see only one header for each constructor since they contain at most one alternative for each constructor and that alternative binds all components of the constructor. Hence only one combination of component representations can occur for each constructor, making the header unique. Note that this technique extends to unboxed tuples as well, in a way similar to *flexible representation analysis* [18].

Another approach to unboxed values in polymorphic data structures is used by Hall, Peyton Jones and Sansom in [7] where they generate specialized monomorphic and unboxed versions of algebraic data types. That transformation is not automatic since it changes the semantics of the program (by making it stricter); it is up to the programmer to decide where to use it.

4.1 Measurements

We have made preliminary measurements of the effectiveness of the various coercion placement strategies. We have used a set of small test programs (the largest is ≈ 700 lines of code). The small size of the programs makes any claims based on these experiments very tentative. With this caveat, we present the programs:

nqh The N-Queens program written with higher order functions wherever possible (`append` is defined in terms of `foldr` etc), run with input 12.
q1 Same as `nqh`, but with all higher order functions manually removed by specialization, run with input 12.
qf Like `q1`, but with deforestation (manually) applied, run with input 12.
factor Factorizes the number 1073602561 into primes in a very inefficient way.
sort Sorts a list of 8000 integers using insertion sort.
leroy A program similar to the Church integer benchmark in [12], maps `quad quad (λx.x+1)`, where `quad = double double` and `double f x = f (f x)`, over the list [1,2, ..., 40000].
event A small event driven simulator from Hartel and Langendoens benchmark suite [9], run with input 400000.
infer A polymorphic type checker from the same source, run with input 1200.
sched A job scheduler, also from that benchmark suite, run with input 14.

We have measured operation counts using instrumentation code emitted by the compiler, instruction counts using the `icount` analyzer included in the Shade distribution and execution time as the user time component reported by the `/bin/time` command. Both times and instruction counts include garbage collection. All measurements were performed on a lightly loaded Sun Ultra 5 workstation with 128MB of memory and a 270MHz UltraIIi processor.

Table 1 compares the performance under the various strategies with the performance when only local optimizations were made. More specifically, coercion

placement in the local case used the Endpoints strategy with the additional constraint that all formal parameters and all variables bound in case alternatives must be boxed.

We have also compiled the code using *cloning* (generating several versions of the same function). Our implementation clones callers before callees which generates more copies than cloning callees first; the result is a program where every function is called non-recursively from a single call site. Clones which are identical after optimization are shared [6].

Table 1. Operation counts and reductions for lazy programs

Prog	Local		Endpoints				Box sharing				No rebox				No cycles				Any			
	B	U	B	U	I	T	B	U	I	T	B	U	I	T	B	U	I	T	B	U	I	T
No cloning																						
`nqh`	9	37	26	63	87	87	26	21	86	84	26	7	84	85	26	7	84	85	26	7	84	85
`q1`	14	55	0	0	72	62	0	0	72	62	0	0	72	62	0	0	72	62	0	0	72	62
`qf`	20	72	0	0	61	41	0	0	61	40	0	0	61	41	0	0	61	41	0	0	61	41
`factor`	0	20	0	0	98	94	0	0	98	100	0	0	98	100	0	0	98	94	0	0	98	100
`sort`	0	21	26	97	97	94	13	96	97	94	2	96	97	96	2	96	97	97	2	96	97	97
`leroy`	11	11	100	99	100	100	100	99	100	100	100	98	99	99	100	98	99	99	100	98	99	99
`event`	8	13	92	80	98	99	0	3	85	80	0	3	85	83	0	3	85	83	0	3	85	83
`infer`	1	14	78	85	99	98	53	83	99	97	97	81	100	99	97	81	100	99	97	81	100	99
`sched`	1	13	94	58	99	98	52	42	98	96	53	34	98	96	53	34	98	96	53	34	98	95
With cloning																						
`nqh`	11	44	0	0	78	77	0	0	78	70	0	0	78	73	0	0	78	73	0	0	78	73
`q1`	14	55	0	0	72	62	0	0	72	62	0	0	72	62	0	0	72	62	0	0	72	62
`qf`	20	72	0	0	61	40	0	0	61	40	0	0	61	40	0	0	61	40	0	0	61	40
`factor`	0	20	0	0	98	100	0	0	98	94	0	0	98	94	0	0	98	94	0	0	98	100
`sort`	0	21	26	97	97	96	13	96	97	96	2	96	97	100	2	96	97	100	2	96	97	100
`leroy`	14	14	100	99	100	99	100	99	100	102	100	98	99	102	100	98	99	101	100	98	99	101
`event`	9	16	0	2	82	75	0	2	82	75	0	2	82	76	0	2	82	75	0	2	82	75
`infer`	1	16	0	0	97	94	0	0	97	93	0	0	97	93	0	0	97	93	0	0	97	93
`sched`	1	14	52	54	98	96	52	42	98	98	53	34	98	95	53	34	98	95	53	34	98	95

The limited reduction in instruction counts of the `infer` and `sched` programs is due to the small part of the instructions that are devoted to executing coercions. We can estimate that number by noting that a `box_int` takes about nine instructions and an `unbox_int` takes one instruction. Using these figures and the columns labeled Local in Table 1, we can estimate that these programs spend about 2% their executed instructions on coercions, in contrast to 25% for `qf`.

It is interesting to compare the cloned and uncloned versions of the same programs. When global coercion placement is used, we see that as we move from Endpoints to Any, the speedup is better for the cloned versions for all strategies,

but the difference is greatest for Endpoints (because almost all coercions are eliminated by Endpoints for the cloned versions of `event` and `nqh`). For the more powerful strategies, the gap narrows. This tells us that cloning pays off, making the Endpoints strategy sufficient, while more flexible placement of coercions can to some extent compensate for the absence of cloning.

Cloning is important for compiling large programs well; an often used function like `append` is applied in so many places that it is unlikely in the extreme that it will be used only with lists where the elements may have the same unboxed representation. Thus it will certainly be compiled to work with lists of boxed objects. This means that all call sites have to pass lists of boxed objects to and from `append` and so on, a problem known as *representation pollution*.

Coercion placement also seems to interact well with other optimizations (specialization of higher order functions and deforestation); the effectiveness of coercion placement increases when these optimizations are applied.

Looking at all of the programs together, it seems not only that the programs which spend most time on coercions show the greatest speedups; it also seems that they have the greatest part of their coercions removed. This may be because the programs which spend much time on coercions are those that spend much time on arithmetic, and strictness analysers are typically more effective for such programs. Also, arithmetic operations are cheap which makes them candidates for the cheap eagerness optimization (speculative evaluation), which further reduces the number of thunks built, and thunks are one of the main obstacles for coercion placement.

5 Related Work

For space reasons, we can only give some brief comments on a few related efforts, all in the context of functional languages. Most work in this area has been done for strict languages where the type of a variable is sufficient to determine its legal representations. The ordinary type information computed by the compiler has been used by Leroy [12] where every variable with a suitable monomorphic type (*e.g.* `Int`) is unboxed, but variables with polymorphic type, such as the formal parameter of the identity function, are boxed. This method can lead to a large amount of unboxing and reboxing (he reports results ranging from a speedup of about 3.5 to a slowdown of about 2.7) and this effect can even change the asymptotic complexity of a program [13].

Most authors [12,19,17] report larger (positive or negative) changes in performance than we have found. We believe that this is due to a combination of factors. In contrast to [17], we use an efficient generational garbage collector which makes the allocation and reclamation of boxes reasonably cheap and our language is typed, so unboxing does not eliminate type tests. We also differ from all of the above by working in a lazy language, making boxing more difficult to eliminate, and by not unboxing floating point numbers (which we don't have) and tuples. For the SML compilers [12,19] this is an especially important optimization since SML uses tuples to pass multiple arguments to functions.

Our use of cloning has a parallel in [2] where intersection types are used to allow improved (closureless) representations of functions.

For lazy languages, both the Glasgow Haskell Compiler [16] and the GRIN compiler [1] use more local techniques where boxings are pushed forwards and unboxings backwards in the hope that they will meet and cancel out; neither compiler pushes coercions through data structures, however.

6 Conclusions and Further Work

This paper presents the second fully automatic global representation analysis for lazy functional programs (the first one is [3]). The problem is divided into finding the set of all legal representation assignments and then selecting the best one; several strategies for the second problem are discussed.

Preliminary measurements on a set of small test programs showed anything from no performance improvement at all (for programs spending negligible time on coercions or where coercions could not be eliminated) to a speedup of about a factor of two for a program where all coercions were eliminated.

It is clearly beneficial to be able to place coercions not only at producers and consumers of data, but no placement strategy was uniformly superior to the others. An interesting extension would be to use profiling data to concentrate on eliminating the most frequently executed coercions.

There are two major limitations to coercion placement as presented in this paper: The current formulation is not compositional, so separate compilation is not possible. Our technique is however directly applicable to whole-program compilers like GRIN [1] and MLton [20]. We also have only dealt with unboxed numbers so far. It is also useful to represent tuples in unboxed form, that is, as a set of variables, one for each component of the tuple. It appears possible, though not trivial, to extend coercion placement in this direction. Since the technique appears otherwise promising, we plan to address these problems in the future.

References

1. Urban Boquist. *Code Optimization Techniques for Lazy Functional Languages.* PhD thesis, Chalmers University of Technology, April 1999.
2. Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. *ACM SIGPLAN Notices*, 32(8), August 1997.
3. Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In Allan Mycroft, editor, *Proceedings of the Second International Symposium on Static Analysis*, pages 136–153, Glasgow, UK, September 1995. Springer-Verlag.
4. Karl-Filip Faxén. *Analysing, Transforming and Compiling Lazy Functional Programs.* PhD thesis, Department of Teleinformatics, Royal Institute of Technology, June 1997.
5. Karl-Filip Faxén. Cheap eagerness: Speculative evaluation in a lazy functional language. In Philip Wadler, editor, *Proceedings of the 2000 International Conference on Functional Programming*, September 2000.
6. Karl-Filip Faxén. The costs and benefits of cloning in a lazy functional language. In Stephen Gilmore, editor, *Trends in Functional Programming, volume 2*, pages 1–12. Intellect, 2001. Proc. of Scottish Functional Programming Workshop, 2000.

7. Cordelia V. Hall, Simon L. Peyton Jones, and Patrick M. Sansom. Unboxing using specialisation. In *Glasgow functional programming workshop*. Springer Verlag, July 1994.
8. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Principles of Programming Languages*, San Francisco, January 1995.
9. Pieter Hartel and Koen Langendoen. Benchmarking implementations of lazy functional languages. In *Functional Programming & Computer Architecture*, pages 341–349, Copenhagen, June 93.
10. Fritz Henglein. Global tagging optimization by type inference. In *Proc. 1992 ACM Conf. on LISP and Functional Programming (LFP), San Francisco, California*, pages 205–215. ACM, ACM Press, June 1992.
11. Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 213–226, Portland, Oregon, January 1994.
12. Xavier Leroy. Unboxed objects and polymorphic typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, Albequerque, New Mexico, January 1992.
13. Yasuhiko Minamide and Jacques Garrigue. On the runtime complexity of type-directed unboxing. In *Proc. International Conference on Functional Programming*, pages 1–12. ACM Press, September 1998.
14. Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, University of Copenhagen, Denmark, January 1997.
15. John Peterson. Untagged data in tagged environments: Choosing optimal representations at compile time. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 89–99. ACM Press, September 1989.
16. Simon L Peyton Jones and André Santos. A transformation-based optimizer for Haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998.
17. Manuel Serrano and Marc Feeley. Storage use analysis and its applications. *ACM SIGPLAN Notices*, 31(6):50–61, June 1996.
18. Zhong Shao. Flexible representation analysis. *ACM SIGPLAN Notices*, 32(8):85–98, August 1997.
19. Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Programming Language Design and Implementation*, pages 116–129, La Jolla, CA, June 1995. ACM Press.
20. Stephen T. Weeks. MLton user's guide. Available from `www.sourcelight.com/MLton`, September 2000.

Pipeline Modeling for Timing Analysis*

Marc Langenbach[1], Stephan Thesing[2], and Reinhold Heckmann[3]

[1] Saarland University,
thesing@cs.uni-sb.de
[2] Saarland University,
mlangen@cs.uni-sb.de
[3] AbsInt Angewandte Informatik GmbH,
heckmann@AbsInt.com

Abstract. In *hard* real-time systems, the worst-case execution times of programs must be known. Obtaining *safe* upper bounds for these times by measuring actual executions is rarely possible, since the worst case input is normally not known. We apply static program analysis methods to determine an upper bound for the WCET. While this approach is not new, we believe to be the first to have developed a tool that implements these techniques for *all* the features of a real-life, non-trivial processor, the Motorola ColdFire 5307. Our tool is, to the best of our knowledge, the first one that can determine a *safe* and rather precise WCET bound for a processor that has caches and pipelines and performs branch prediction and instruction prefetching.
Our approach to use a pipeline model in the analysis of the processor behavior opens up new perspectives towards a generative analysis approach and can prove helpful in investigating other processor properties. The emphasis of this paper is on the modeling of the pipeline behavior as input to the derivation of a pipeline analysis.

1 Introduction

Real-time systems are computer systems that interact closely with the physical environment. In such a system correctness depends not only on the logical results, but also on the time at which the results are produced. Failure of a safety-critical (also called *hard*) real-time system can lead to severe damage. Such real-time systems require timing validation by a schedulability analysis. All existing techniques for schedulability analysis require as input the worst-case execution time (WCET) of each task of the system.

In modern microprocessor architectures caches and pipelines are key features for improving performance. Caches are used to bridge the gap between processor speed and the access time of main memory. Pipelines enable acceleration by overlapping executions of several instructions. Caches and pipelines can considerably improve performance but often at the expense of reduced predictability:

* This work was partly supported by the RTD project IST-1999-20527 DAEDALUS of the European FP5 program.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 294–309, 2002.
© Springer-Verlag Berlin Heidelberg 2002

The state of the cache and pipeline depends on the execution history. Therefore, classical approaches [22,24] to WCET prediction are not directly applicable or lead to results overestimating the real execution time by orders of magnitude. At the Ada Deutschland conference, Alfred Roßkopf of EADS presented results showing that a PowerPC 604 running at 300MHz with caches disabled (and thus more predictable) delivers a similar performance on some benchmarks as a Motorola 68020 running at 20MHz, while the PowerPC outperforms the 68020 by a factor of 20 with caching enabled. For applications like fly-by-wire which require a high computing performance, disabling caches is not an option.

In our approach the prediction of WCETs is divided into two phases which are tackled with different methods: *abstract execution modeling* and *program path analysis*. The times to execute program paths are determined using abstract interpretation [4] based on a formal model of the target system. The other phase, program path analysis, is responsible for determining a worst-case execution path. The problem is formulated as an integer linear program (ILP), cf. [27], and solved by *lp_solve*. A more detailed description of the workflow is given in Section 3.

Though we are not the first to present methods for WCET determination for processors with caches and/or pipelines, our tool, we believe, is the first one that models a non-trivial real-life processor, the Motorola ColdFire 5307, taking into account all of its features in combination, like instruction prefetching, branch prediction, caches, the pipeline, and the memory bus. Other tools known to the authors (such as [16], [20], [3]) implement some of the features listed above, but make assumptions that lead to unusuable results in the context of hard real-time systems.

We applied our tool to a real-life benchmark provided by AirBus France in the DAEDALUS project[1], containing realistically sized code modules.

The modeling and implementation of our tool showed that some assumptions or simplifications made by other WCET methods are not valid for the ColdFire, e. g. that a cache-miss always leads to longer execution times than a cache-hit. Indeed, there are cases where a cache-hit results in a higher overall execution time of the program although it is locally faster, cf, section 4.3.

In this paper, we focus on the pipeline model that was used to implement the static pipeline behavior analysis. In using a model to capture the pipeline behavior, we can open up the perspective of generating the pipeline analysis from the abstract model.

In Section 2, we give an overview of work related to this article. Section 3 describes the components of our tool. In Section 4, we present the ColdFire pipeline and our formal model. Section 5 contains some practical results and conclusions on our work.

[1] The DAEDALUS project aims at introducing static program analysis methods into the airplane software validation process. Partners include AirBus France, AbsInt, PolySpace and various academic partners, e.g. ENS, Saarland University, etc.

2 Related Work

2.1 WCET Determination

Based on Shaw's timing scheme [24] which focuses on high-level language constructs and does not consider cache or pipeline effects, Lim et al. present in [16, 17] and Hur et al. in [13] an analysis technique accounting for modern processor features. Cache effects are modeled via bookkeeping of first and last references to blocks and reservation tables are used to handle pipeline effects. As the target machine—a MIPS 3000—implements only a simple pipeline, reservation tables whose resources are registers and pipeline stages are sufficient. Results were only reported for toy sample programs. More sophisticated processors featuring out-of-order execution, superscalarity, or set-associative caches are not considered.

Healy et al. [9,10] present another approach on predicting WCETs in the presence of caches and simple pipelines. In a first step of the analysis a static cache simulator classifies instructions as cache hits or misses. This information is used by a pipeline path analysis that computes the execution time for a sequence of instructions. Loops are handled in a bottom-up manner. Only the simple pipeline of a MicroSPARC is considered and in [9] only direct-mapped caches and simple pipelines are taken into account which can be described by resource usage patterns of instructions. For their experimental results the authors only consider a small direct-mapped cache with small test programs.

Li et al. suggest a solution using integer linear programming [15]. Both cache and pipeline behavior prediction are formulated as one linear program. The i960KB is investigated which has a fairly simple pipeline. So only structural hazards need to be modeled keeping the complexity of the integer linear program moderate. Branch prediction and/or instruction prefetching are not considered at all. Obtaining the ILP modeling for a more complex processor will be difficult. Using their approach for superscalar pipelines does not seem very promising considering the analysis times reported in the article. Nonetheless, the description of the worst-case path through the program via ILP is an elegant method and can be efficient if the size is kept small as is the case in our tool.

Lundqvist and Stenström present an integrated approach to obtain WCET bounds through simulation of the pipeline in [18,19]. They extend a pipeline simulator to handle unknown values in inputs. With this approach we share conceptual similarities in that we perform a cycle-wise evolution of a pipeline (model). Different from our approach their method is an integrated one, where value analysis for register/memory contents and execution time computation are parts of the same simulation. If the simulation cannot determine a branch condition exactly due to dependencies on unknown (input) values, they have to simulate both branches. Their method does not guarantee termination of the analysis but has the advantage of determining loop bounds and/or recursion bounds for free[2]. However, we feel that their analysis is very costly due to the huge amount of data that has to be kept for each branch they follow. In contrast,

[2] If these do not depend in a non-trivial way on unknown input values.

our method does not keep information like register or memory contents in the pipeline analysis phase. A value analysis can be executed before the cache and pipeline analysis instead. In [19] experiments with a PowerPC-like architecture are made for small example programs. They use an extended PSIM simulator with simple reservation tables for instructions. In all, it is not clear how well their method scales up to programs of realistic sizes.

Narasimhan and Nilsen present a retargetable execution time analyzer for RISC processors in [20]. The target architecture is modeled using an extended MARIL language (see [2]). The generated analyzer takes an assembly program and a path represented by a sequence of labels and computes the time needed to execute that path. Due to the use of MARIL the range of targetable processors is significantly limited. Analyzing assembly programs complicates the integration of instruction and data cache analysis. This leads to a large gap between predicted and measured execution time, especially for larger inputs [20].

In [3] Colin and Puaut present a method to analyze dynamic branch prediction by obtaining classifications for branches and computing an upper bound on the number of mispredictions. The method can only be applied if the input programs adhere to rather strict syntactic constraints, questioning the applicability to real-life examples.

In [5] Ferdinand presents a method for static cache and pipeline analysis based on abstract interpretation. Using this methodology, Schneider et al. developed a pipeline behavior prediction [23] for the SuperSPARC processor. Both analyses are components of a worst case execution time prediction. However, their approach uses a simple reservation table scheme in the pipeline analysis which cannot be extended to more complex architectures. Also, they do not consider prefetching and/or branch prediction and assume the sequence of instruction accesses to be known statically. Ferdinand's cache analysis is integrated into our pipeline analysis, adapted to the ColdFire's cache update semantics.

None of the work mentioned above takes into account the combination of branch prediction, instruction prefetching, speculative execution, or effects caused by, e. g., data accesses colliding with wrap-around cache line fills on the external processor bus. Unfortunately, these features are not orthogonal, but influence each other. Therefore, they *must* be considered together to obtain tight WCET predictions.

Also, modern computers feature separated core and bus clocks, causing new wait effects for external bus accesses, which have not been investigated so far. Most models simplify the memory architecture of the systems they consider; in a real system, the memory space is made up of quite a few regions with different characteristics concerning access timing, access mode[3], and so on. Our tool makes it easy to specify these parameters and the pipeline analysis takes all this into account when examining memory accesses.

[3] E. g., speculative accesses with the PowerPC can be allowed on some regions but disallowed on others.

2.2 Pipeline Modeling

Apart from retargetable timing analysis, pipeline descriptions have been used in code generation and simulation. Early work [2] considered only registers and pipeline stages by the use of reservation tables. Improvements have led to mixed-level languages such as Expression [8]. From a structural specification of hardware resources, pipeline mechanism, and data transfer paths, reservation tables are generated automatically [7], but this approach is not able to model dynamic behavior like out-of-order execution.

Dynamic hardware features are handled in hardware description languages that are used for retargetable simulators. The language LISA [28,21] uses a refined form of reservation tables called L-charts that can model dynamic scheduling to some extent. The language RADL [25] offers a flexible signal mechanism that seems more promising in the context of modern hardware features. Signals are emitted if a boolean expression on machine state components holds; they are used to influence scheduling behavior. In contrast to our work which also deals with signals, RADL describes a model very close to the silicon level, where, e. g., latches between pipeline stages are defined implicitly and may be accessed bitwise.

3 The ColdFire WCET Tool

The operation of our WCET tool can be roughly divided into two subsequent phases: *execution modeling* and *program path analysis*. In the first phase we use an analyzer to determine the time needed to execute a program path. In the second stage, a worst-case execution path is identified.

Both phases work on a representation of the control flow graph (CFG). This intermediate representation called CRL is generated from an executable. Analyzing an executable has several advantages. After linking, addresses and sizes of instructions are known which is essential for the cache and pipeline analysis. There are no undefined call targets[4] as their definitions have been added during linking. Another big advantage of analyzing executables is that the toolchain for program development must not be changed to introduce new components or modify existing ones[5]. In the airplane industry, e.g., the requirements imposed by the certification authorities for development tools are very strict. Here, a modified toolchain has to be certified again for development, which may be a very time-intensive and costly procedure.

The component that extracts the CFG is called *exec2crl* [26] and can rather easily be adapted to different executable formats/compilers and architectures.

The execution modeling itself is composed of several analyses. First, a value analysis collects information about the contents of registers and accessed memory cells. This information helps to determine the effects of indirect memory accesses

[4] Function pointers are handled by the value analysis.

[5] Some of the approaches mentioned in Section 2 rely on a modified compiler to extract additional information.

on the cache. It is also used to recognize infeasible paths, which improves the results of the following analyses.

A combined cache and pipeline analysis is started next, statically determining all possible combinations of pipeline states and cache states that might occur during program execution. This analysis is presented in more detail in the next section.

While cache analysis could have been performed separately from and before pipeline analysis, this would have required an additional analysis to determine an approximation to the instructions that can be accessed by the program due to the presence of instruction prefetching and branch prediction in the ColdFire, which is dependent on the pipeline execution. But since this would lead to a decrease of precision in the cache analysis and because of other analysis drawbacks caused by the design of the ColdFire cache itself, we integrated these analyses to gain maximum precision.

Value analysis and combined cache and pipeline analysis are based on abstract interpretation and implemented as data-flow analyses using the program analyzer generator **PAG** [1].

To determine the WCET of the input program, the path analysis takes the results of the preceding analyses along with some optional user provided information on loop counts and execution constraints and formulates the problem of finding a longest execution path as an integer linear program (ILP). Solving the ILP using, e. g., `lp_solve` or CPLEX yields the desired upper bound for the WCET. The results can then be visualized with the graph layout tool **aiSee** [12] and interactively explored. This gives the opportunity to see the WCET contributions of all functions and even all instructions in the CFG. Another option is to visualize the evolution of the pipeline cycle by cycle so that one can reproduce the results on any level of abstraction.

4 Pipeline Analysis

The purpose of the combined cache and pipeline analysis is to give an upper bound of the WCET for each *basic block* in the CFG. We do this by statically simulating the execution of the instructions in the CFG.

Since our method is based on abstract interpretation, we start with a concrete semantics of the instruction execution and then derive an abstraction that is implemented as a data-flow analysis in the CFG of the program.

The concrete semantics uses a *model* of the pipeline, in which only the effects that have an impact on the execution timing are present. For instance we are only interested in how long a multiplication instruction takes, but not what its arguments or results are[6].

The nodes of our CFG are individual machine instructions. Since the execution of some instructions (branches, etc.) depends on their successor in the CFG,

[6] On the ColdFire 5307 a multiplication always takes the same time. This is not true on, e. g., the PowerPC 755, where the time for a multiplication depends on the operands.

we associate the transfer functions of the analysis with the edges between nodes. The analyzer moves along an edge e between two nodes (and thus instructions) v and v' when execution of v has completed. This statement may seem trivial, but since the execution of machine instructions is distributed among several pipeline stages and thus several instructions are being executed at the same time, the meaning of the CFG must be clearly defined. As another alternative, one can pass along an edge between v and v' as soon as the execution of v *begins*. However, we choose the first alternative since it is simpler in the presence of branch prediction. Hence, the transfer function of an edge must simulate the effects on the pipeline (model) until the instruction at the source node of the edge has completed execution.

The system state in the concrete semantics is made up of the cache contents and the pipeline state. When abstracting, we use separate abstractions for the cache contents and the pipeline state.

The abstract cache state follows the semantics presented in [5], adopted to the cache replacement strategy used by the ColdFire 5307. There, an abstract cache state can represent memory blocks that are guaranteed to be in the cache (*must* information) or cache blocks that may be in the cache (*may* information). From this, classifications of memory accesses as guaranteed hits or misses (or unknown) can be derived. The abstraction that can be found for the ColdFire cache, which is a 8KB, 4-way associative unified instruction/data cache with a 'pseudo round-robin' replacement [14] suffers badly from some worst-case features of the hardware. In essence, we use an abstract cache model equivalent to the *must* information of a 2KB direct mapped cache, cf. [6,11]. This means that we can never classify memory accesses as guaranteed cache misses. Also, the (virtual) size reduction of the cache reduces the amount of guaranteed cache hits, which has the potential to lead to a very pessimistic analysis.

In order to introduce the pipeline model, we give an overview of the ColdFire 5307 in the next section, followed by the model and the abstraction.

4.1 The ColdFire MCF 5307 Pipeline

The ColdFire 5307 is a successor to the well known M68K architecture, inheriting most of its instructions. Some restrictions on instruction formats and sizes have been made to enable a fast implementation. The MCF 5307 features a set of integrated peripherals, like DMA engines and serial ports. It is a very popular microcontroller used in many applications, e.g. the AirBus planes.

The pipeline of the ColdFire MCF 5307 consists of a *fetch pipeline* where instructions are fetched from memory (or the cache), and an *execution pipeline* where instructions are executed (cf. Figure 1). Fetch and execution pipeline are decoupled by a FIFO instruction buffer that can hold at most 8 instructions.

The MCF 5307 has a bus hierarchy to access memory. The pipeline is connected to the K-Bus, a fast pipelined bus that connects the cache and an internal 4KB SRAM area to the pipeline. Accesses to this bus are performed by the IC1/IC2 and the AGEX and DSOC stages. On the next level, the M-Bus the internal peripherals are connected. This bus runs at the external bus frequency,

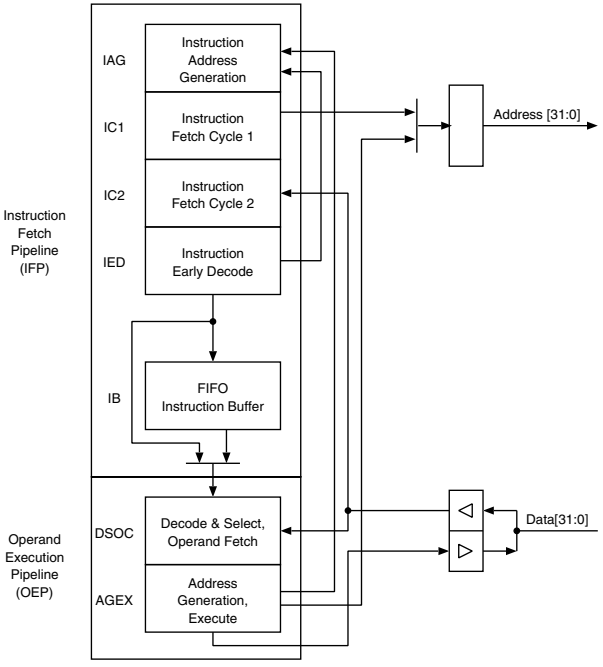


Fig. 1. The Coldfire 5307 Pipeline

while the K-Bus is clocked with the faster internal core clock. The M-Bus, connects to the external bus, which accesses off-chip peripherals and memory.

The ColdFire cache is a non-blocking wrap-around line-fill cache, which means that on a cache miss the requested longword is fetched first from external memory and delivered to the pipeline, followed by the remaining three words from that line.

The fetch pipeline performs *branch prediction* in the IED stage, redirecting fetching long before the branch reaches the execution stages. The fetch pipeline must be stalled if the instruction buffer is full, or if the execution pipeline needs the bus for a memory access. All these stalls cause the pipeline to wait for one cycle, then the stall condition is checked again.

The fetch pipeline is also stalled if the memory block to be fetched is not in the cache (cache miss). In this case, the pipeline must wait until the memory block is loaded into the cache and forwarded into the pipeline. The instructions that are already in the later stages of the fetch pipeline are forwarded to the instruction buffer.

The execution pipeline finishes the decoding of instructions, evaluates their operands, and executes the instructions.

We assume that each kind of operation follows a fixed schedule, telling how many cycles it needs and in which cycles memory is accessed[7]. The execution time varies between 2 cycles and several dozen cycles. Pipelining admits a maximum overlap of 1 cycle between consecutive instructions: the last cycle of each instruction may overlap with the first of the next one. In this first cycle, nothing important happens (no memory access, and no control-flow alteration). Thus, cache and pipeline cannot be affected by two different instructions in the same cycle.

The execution of an instruction is delayed if memory accesses lead to cache misses. Misaligned accesses lead to small time penalties of 1–3 cycles. Store operations are delayed if the distance to the previous store operation is less than 2 cycles. (This does not hold if the previous store operation was issued by a `MOVEM` instruction.) The start of the next instruction is delayed if the instruction buffer is empty.

4.2 A Formal Pipeline Model

On a detailed level, the processor pipeline can be described as a big state machines with clock-cycle wise transitions from pipeline state to pipeline state. A pipeline state contains all timing relevant information of the processor and the number of transitions from the beginning of the execution of an instruction until its end gives the execution time of that instruction. The transitions between states and the states themselve can be very complicated, and writing them down results in a complicated construct, difficult to handle and to implement.

In our formal model introduced here, a concrete pipeline state consists of several *units* with inner *states* that communicate with one another and the memory via *signals*, and evolve cycle-wise according to their inner state and the signals received. Thus, the big flat pipeline state is divided into (disjoint) portions that are encapsulated into units.

The decomposition into units accounts for reduced complexity and easier validation of the model. Units often map directly to pipeline stages, but also may represent more than one stage or introduce virtual pipeline stages that are not present in hardware but facilitate the design of the pipeline model (cf. the store stall timer).

Signals may be *instantaneous*, meaning that they are received in the same cycle as they are sent, or *delayed*, meaning that they are received one cycle after they have been sent. Signals may carry data, e.g. a fetch address. Note that these signals are only part of the formal pipeline model. They may or may not correspond to real hardware signals.

By seperating the complex state into smaller (disjoint) units, complexity is reduced and implementation is made easier. The instantaneous signals between units are now used to transport information between the corresponding portions

[7] In fact, there are some instructions like `MOVEM` whose execution schedule depends on the value of an argument given as immediate constant. These instructions can be handled by special means.

of the flatted out state. The state transitions are coded in the evolution rules local to each unit.

Figure 2 shows the formal pipeline model for the ColdFire MCF 5307. The following units exist: IAG (instruction address generation), IC1 (instruction fetch cycle 1), IC2 (instruction fetch cycle 2), IED (instruction early decode), IB (instruction buffer), EX (execution unit), SST (store stall timer). In addition, there is a *bus unit* modeling the busses that connect the CPU, the static RAM, the cache, and the main memory. The signals between these units are shown as arrows. Most units directly correspond to a stage in the real pipeline, but the SST unit is used to model the fact that two stores must be seperated by at least two clock cycles by implementing a (virtual) counter. Furthermore, the two stages of the execution pipeline are modeled by a single stage, EX, because instructions can only overlap by one cycle.

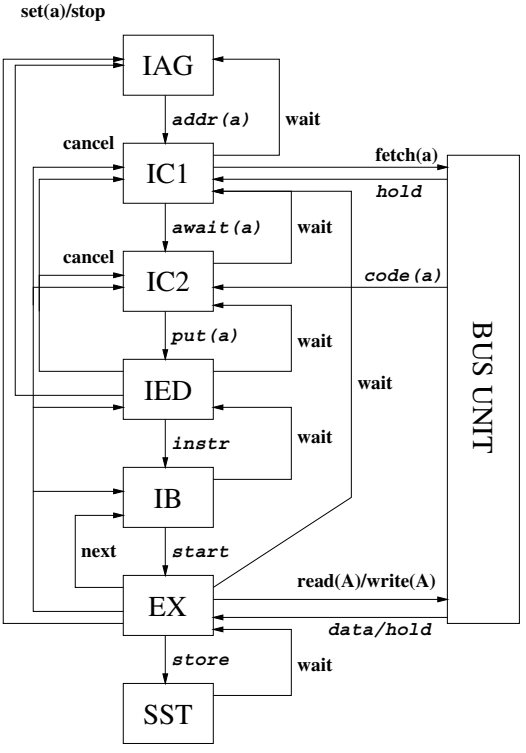


Fig. 2. Map of the formal pipeline model

The inner states and emitted signals of the units evolve in each cycle. The complexity of this state update varies from unit to unit. It can be as simple as a small table, mapping pending signals and inner state to a new state and

signals to be emitted, e.g. for the IAG unit. It can be much more complicated, if multiple dependencies have to be considered, e.g. the instruction reconstruction and branch prediction in the IED stage. In this case, the evolution is formulated in pseudo code.

Full details on the model can be found in [11].

4.3 Pipeline States

Concrete Pipeline States are formed by combining the inner states of IAG, IC1, IC2, IED, IB, EX, SST, and bus unit plus additional entries for pending signals into one big state. This big state evolves from one cycle to the next. Practically, the evolution of the big pipeline state can be implemented by updating the functional units one by one in an order that respects the dependencies introduced by input signals and the generation of these signals.

Update Function for Pipeline States. For pipeline modeling, one needs a function that describes the evolution of the concrete pipeline state while traveling along an edge (v, v') of the control-flow graph. This function can be obtained by iterating the cycle-wise update function of the previous paragraph.

A concrete pipeline state at v has an empty execution unit EX. It is updated until an instruction is sent from IB to EX. Updating of the concrete pipeline state continues using the knowledge that the successor instruction is v' until EX has become empty again. The number of cycles needed from the beginning until this point can be taken as the time needed for the transition from v to v' for this concrete pipeline state.

Abstract Pipeline States. The abstraction we use for pipeline states is just a *set* of concrete pipeline states, since the amount of possible pipeline states for one instruction tends to be tolerable. Hence, our abstraction computes an upper bound to the collecting semantics.

When considering the abstract update of abstract pipeline states, i.e. the data flow analysis with the transfer functions at the edges of the CFG, it can no longer be assumed that an update is *deterministically*, as has been assumed in the concrete pipeline update. Consider for instance indirect load and store operations, where the address that is read or written is not known in the analysis. Accessing an unknown memory address leads to a cache update with unknown address, destroying the contents of the cache state. The time until a read request to an unknown address is served may vary wildly depending on whether a cache hit or cache miss results and on the memory regions present in the system (cacheable, uncacheable regions, peripherals, etc.). To accommodate all cases, the concrete pipeline state is split into two or more successor states, which must be interpreted independently in the sequel. The number of successor states is determined by the number of memory regions in the system that can be the target of the load/store.

The abstract update for an abstract pipeline state is thus the application of the concrete update on each concrete pipeline state in the abstract state, extended with the possibility of multiple successor states in case of uncertainties.

One may wonder why we do not discard the cache hit successor in the event of an uncertain cache access and only follow the cache miss case. This is because we found out that a cache miss is *not always* the worst case alternative. Since the MCF 5307 has a unified cache and the fetch and execute pipelines are independent, the following can happen: a data access that is a cache hit is served directly from the cache. At the same time, the fetch pipeline fetches another instruction block from main memory, performing branch prediction and replacing two lines of *data* in the cache. These may be reused later on and cause two misses. If the data access was a cache miss, the instruction fetch pipeline may not have fetched those two lines, because the execution pipeline may have resolved a misprediction before those lines were fetched. On certain occasions one can argue that certain states will not produce a higher execution time than others. But because of the parallel workings of the pipelines and the interactions caused by deferring bus accesses due to core and bus clocks running at different speeds (requiring waits for clock alignment), it is not easy to see (and prove!) that some states are worse than others. But we can always remain on the safe side by considering all pipeline states.

Integrated Cache and Pipeline Analysis. When combining the abstract cache and pipeline state to one big abstract state for the analysis, one has two choices: there can be one abstract cache state per abstract pipeline state, i.e. one abstract cache state representing the cache contents for all concrete pipeline states in the abstract pipeline state, or there can be one abstract cache state per concrete pipeline state.

The first choice saves memory during the analysis but loses precision, since the different concrete pipeline states may cause different memory accesses and thus cache contents, which have to be merged into the one abstract state, losing information.

The second choice is more precise but requires more memory during the analysis. In our implementation, we selected the second option.

The update function belonging to an edge (v, v') of the control-flow graph updates each concrete pipeline state separately. When the bus unit is updated, the pipeline state may split into several successor states with different cache states.

The initial state at the beginning of the analysis is a set of empty pipeline states plus an empty cache state[8]. There must be multiple concrete pipeline states in the initial states, since the adjustment of internal to external clock of the processor is not known in the beginning and every possibility (aligned, one cycle apart, etc) has to be considered. Thus prefetching must start from scratch, but pending bus requests are ignored. To obtain correct results, they

[8] Which represents a cache with unknown content.

must be taken into account by adding a fixed penalty to the calculated worst-case execution time.

We should note that our analysis framework allows us to not only perform traditional intraprocedural data flow analysis but to perform interprocedural and context sensitive analyses. E.g. we can differentiate the iterations of loops by introducing contexts representing the first, second, etc. iteration of the loop. To increase precision, we virtually unroll the loops[9] and analyze each iteration on its own.

5 Results and Conclusions

We implemented the presented pipeline analysis in our WCET tool for the ColdFire MCF 5307. The implementation uses the program analyzer generator PAG [1]. Non-determinism in the analysis only occurs during updates of the bus unit, if memory accesses cannot precisely be classified as cache hits or if the access address is not precisely known.

Our tool was tested with a specific hard real-time benchmark which has been developed and provided by AirBus. This benchmark is designed in a way to resemble "real" avionics software. It basically consists of the so-called cyclic executable that reacts to external events by triggering one of 12 independent tasks. Each single task has to satisfy timing constraints.

Analysis times for all of the tasks in the benchmark are below 30 minutes on a 1.0GHz Athlon running RedHat Linux with 1.5GB memory. The benchmark itself is under a non-dislosure agreement with AirBus. Characteristics of the benchmark, analysis results and a comparision against WCET results obtained by other methods will be presented at the conference. However, our method compares very well to the ones used by AirBus and accepted by the airworthiness authorities.

Applying our method to toy programs as used by other groups, the predicted WCET differs from measured execution times by a factor of up to two. Compared to the results of other groups that can only handle rather simple architectures, this factor might seem high. But those results were obtained on "analysis-friendly" hardware, which is rarely used in real-life systems. Our method has the advantage that it scales up to real-life sized programs.

Our WCET prediction tool for the Motorola ColdFire MCF5307 processor has been installed in AirBus Toulouse plant. The initial assessment AirBus software verification specialists carried out was positive. AirBus is currently in the process to decide to use the WCET technology for the timing validation of avionics software (now evolving and future aircraft programs).

During the evaluation of the analysis results, we came across several interesting timing effects, e.g. in one loop the first three loop iterations all had widely varying execution times before the execution time stayed the same for the remaining iterations. By following the cycle-wise pipeline evolution, we discovered

[9] Annotations about the number of loop iterations have to be given anyhow for the path analysis, so we can use them here, too.

that instruction prefetching and instruction/data cache interference were responsible for this effect.

Our approach of separating the pipeline components into smaller units with inner state and the notion of communication via instantaneous and delayed signals simplified the implementation of a non-trivial pipeline.

For the first time, a WCET tool models instruction prefetching and branch prediction together with memory access effects in the bus unit and write buffers while still considering all details and features of an off-the-shelf processor core, the MCF 5307, that are of importance for the WCET determination.

Despite the precise and complex modeling we were able to analyze a real-life benchmark with good results.

Our technique of using a pipeline model as the basis for pipeline behavior prediction also works for even more complicated architectures. We are currently finishing the pipeline analysis tool for the PowerPC 755, a processor that features out-of-order execution, speculation and superscalarity. Also, our technique opens the perspective to a generative approach, where analyses are generated from the specification of a model.

References

1. Martin Alt and Florian Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *Proceedings of SAS'95, Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
2. David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The Marion System for Retargetable Instruction Scheduling. In Brent Hailpern, editor, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 229–240, Toronto, ON, Canada, June 1991. ACM Press.
3. A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Real-Time Systems, Special issue on worst-case execution time analysis*, 18(2):249–274, 2000.
4. Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York.
5. Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Universität des Saarlandes, 1997.
6. Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of EMSOFT 2001, First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, 2001.
7. Peter Grun, Ashok Halambi, Nikil Dutt, and Alex Nicolau. RTGEN: An Algorithm for Automatic Generation of Reservation Tables from Architectural Descriptions. In *Proceedings on the 12th International Symposium on Systems Synthesis*, 1999.
8. Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *DATE*, 1999.

9. Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.
10. Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.
11. Reinhold Heckmann and Stephan Thesing. Cache and Pipeline Analysis for the ColdFire 5307. Technical report, Universität des Saarlandes, 2001.
12. http://www.aisee.com. *aiSee Home Page*.
13. Yerang Hur, Young Hyun Bae, Sung-Soo Lim, Sung-Kwan Kim, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Minsuk Lee, Heonshik Shin, and Chong Sang Kim. Worst Case Timing Analysis of RISC Processors: R3000/R3100 Case Study. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 308–319, December 1995.
14. Motorola Inc. *MCF5307 ColdFire Integrated Microprocessor User's Manual*. Motorola Inc., August 2000. MCF5307UM/D, Rev. 2.0.
15. Y.-T. S. Li, S. Malik, and A.Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. *IEEE Real-Time Systems Symposium*, January 1997.
16. Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, and Chong Sang Kim. An Accurate Worst Case Timing Analysis Technique for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7), July 1995.
17. Sung-Soo Lim, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, and Chong Sang Kim. Issues of Advanced Architectural Features in the Design of a Timing Tool. In *Proceedings of the 11th IEEE Workshop on Real-time Operating Systems and Software*, 1994.
18. Thomas Lundqvist and Per Stenström. Integrating Path and Timing Analysis Using Instruction-Level Simulation Techniques. In Frank Mueller and Azer Bestavros, editors, *Proceedings of the ACM SIGPLAN Workshop Languages, Compilers and Tools for Embedded Systems (LCTES)*, volume 1474 of *Lecture Notes in Computer Science*, pages 1–15, 1998.
19. Thomas Lundqvist and Per Stenström. An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. *Real-Time Systems Journal*, 17(2/3):183–207, November 1999.
20. K. Narasimhan and K. Nilsen. Portable Execution Time Analysis for RISC Processors. 1994.
21. S. Pees, V. Zivojnovic, A. Hoffmann, and H. Meyr. Retargetable Timed Instruction Set Simulation of Pipelined Processor Architectures. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, September 1998.
22. P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 2(1):159–176, 1989.
23. Jörn Schneider and Christian Ferdinand. Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 34, pages 35–44, May 1999.
24. Alan C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

25. Chuck Siska. A Processor Description Language Supporting Retargetable Multi-Pipeline DSP Program Development Tools. In *Proceedings of the 11th International Symposium on System Synthesis*, 1998.
26. Henrik Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing Systems and Applications*, Cheju Island, South Korea, 2000.
27. Henrik Theiling and Christian Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, 1998.
28. V. Zivojnovic, S. Pees, Ch. Schläger, Markus Willems, Rainer Schoenen, and H. Meyr. LISA – Machine Description Language and Generic Machine Model. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, October 1996.

Analysing Approximate Confinement under Uniform Attacks

Alessandra Di Pierro[1], Chris Hankin[2]*, and Herbert Wiklicky[2]

[1] Dipartimento di Informatica, Universitá di Pisa, Italy
[2] Department of Computing, Imperial College, London, UK

Abstract. We are concerned to give certain guarantees about the security of a system. We identify two kinds of attack: the internally scheduled attack (exemplified by Trojan Horse attacks) and externally scheduled attacks (exemplified by timing attacks). In this paper we focus on the latter. We present a semantic framework for studying such attacks in the context of PCCP, a simple process algebra with a constraint store. We show that a measure of the efficacy of an attacker can be determined by considering its observable behaviour over the "average" store of the system (for some number of steps). We show how to construct an analysis to determine the average store using the technique of probabilistic abstract interpretation.

1 Introduction

Confidentiality is that aspect of computer security concerned with how information is allowed to flow through a computer system. Information flows are treated in a binary fashion: they are either allowed to flow or not. Models for confidentiality typically characterise the absence of information flow between objects (across interfaces or along channels) by essentially reducing non-interference to confinement. The first attempt to formalise such an absence of information flow is the notion of non-interference proposed by Goguen and Meseguer in their seminal paper [12]. In this paper we approach the problem of confidentiality by looking at models which are able to give a *quantitative* estimate of the information flowing through a system. Such models abandon the purely qualitative binary view of the information flow by characterising how much information is actually "leaking" from the system rather than the complete absence of any flow. This allows us to define notions of non-interference which are *approximate* and yet able to capture the security properties of a system in a more "realistic" way: in real systems high-level input interferes with low-level output all the time [16].

Our approach is therefore not to aim for a methodology to develop "secure" systems — e.g. a type system which guarantees the confidentiality of information [20,22,21] — rather our approach is to develop a framework for understanding and analysing systems as they are and not as they should be.

* Chris Hankin is partly funded by the EU FET open project SecSafe.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 310–325, 2002.
© Springer-Verlag Berlin Heidelberg 2002

The security of a system may be measured by its robustness to certain kinds of attack. We differentiate two situations: internally scheduled spies in which the spy is considered part of the system (Trojan Horse attacks are of this kind) – we studied this kind of spy in [8]; externally scheduled spies in which the spy is only allowed to observe the system at fixed points and for a fixed time – this is the subject of this paper. The latter kind of attack is suitable for modelling *timing attacks* [13]; with this kind of attack a snooper can determine a private key by keeping track of *how long* a computer takes to decipher messages.

This paper introduces a framework for studying attacks involving externally scheduled spies. We use a simple probabilistic process calculus, PCCP, to illustrate the main concepts; PCCP is introduced in the next section. We formally define the notion of approximate confinement for both internally scheduled and externally scheduled spies in Section 3. We then present an analysis of approximate confinement for external attack and illustrate the technique with some examples in Section 4.

2 The Language

We illustrate our approach by referring to a probabilistic declarative language, namely Probabilistic Concurrent Constraint Programming (PCCP), which was introduced in [9] as a probabilistic version of the Concurrent Constraint Programming (CCP) paradigm [19]. This language can be seen as a kind of "process algebra" enhanced with a notion of "computational state", referred to as "store". These states are ordered by an entailment relation $\vdash$ (also denoted by $\sqsupseteq$) — they form a cpo — and all computations lead to sequences of stores which are *monotone* with respect to $\vdash$.

The syntax and the basic execution model of PCCP are very similar to CCP. Both languages are based on the notion of a generic *constraint system* $\mathcal{C}$, defined as a cylindric algebraic complete partial order (see [19,5] for more details), which encodes the information ordering. In PCCP probability is introduced via a probabilistic choice which replaces the nondeterministic choice of CCP, and a form of probabilistic parallelism, which replaces the pure nondeterminism in the interleaving semantics of CCP by introducing priorities.

The syntax of a PCCP agent is given by the following grammar, where c and c_i are *finite* constraints in $\mathcal{C}$, and p_i and q_i are real numbers representing probabilities:

$$A ::= \mathbf{stop} \mid \mathbf{tell}(c) \mid \square_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow p_i : A_i \mid \|_{i=1}^{n} q_i : A_i \mid \exists_x A \mid p(x).$$

2.1 Operational Semantics

The operational semantics of PCCP is defined in terms of a probabilistic transition system, $(\mathrm{Conf}, \longrightarrow_p)$, where Conf is the set of configurations $\langle A, d\rangle$ representing the state of the system at a certain moment and the transition relation $\longrightarrow_p$ is defined in Table 1. The state of the system is described by the agent

Table 1. The Transition System for PCCP

R1	$\langle \mathbf{tell}(c), d\rangle \longrightarrow_1 \langle \mathbf{stop}, c \sqcup d\rangle$	
R2	$\langle \Box_{i=1}^{n} \mathbf{ask}(c_i) \to p_i : A_i, d\rangle \longrightarrow_{\tilde{p}_j} \langle A_j, d\rangle$	$j \in [1,n]$ and $d \vdash c_j$
R3	$\dfrac{\langle A_j, c\rangle \longrightarrow_p \langle A'_j, c'\rangle}{\langle \parallel_{i=1}^{n}\ p_i : A_i, c\rangle \longrightarrow_{p\cdot\tilde{p}_j} \langle \parallel_{j\neq i=1}^{n} p_i : A_i \parallel p_j : A'_j, c'\rangle}$	$j \in [1,n]$
R4	$\dfrac{\langle A, d \sqcup \exists_x c\rangle \longrightarrow_p \langle A', d'\rangle}{\left\langle \exists_x^d A, c\right\rangle \longrightarrow_p \left\langle \exists_x^{d'} A', c \sqcup \exists_x d'\right\rangle}$	
R5	$\langle p(y), c\rangle \longrightarrow_1 \langle A, c\rangle$	$p(x) : -A \in P$

A which has still to be executed, and the common store d. The index p in the transition relation indicates the probability of the transition to take place. The rules are closely related to the ones for nondeterministic CCP, and we refer to [5] for a detailed description. The rules for probabilistic choice and prioritised parallelism involve a normalisation process needed to re-distribute the probabilities among those agents A_i which can actually be chosen for execution. Such agents must be enabled (i.e. the corresponding guards $\mathbf{ask}(c_i)$ succeed) or active (i.e. able to make a transition). The probability after normalisation is denoted by $\tilde{p}_j$.

Definition 1. *Let A be a PCCP agent. A computational path π for A in store d is defined by $\pi \equiv \langle A_0, c_0\rangle \longrightarrow_{p_1} \langle A_1, c_1\rangle \longrightarrow_{p_2} \ldots \longrightarrow_{p_n} \langle A_n, c_n\rangle$, where $A_0 = A$, $c_0 = d$, $A_n = \mathbf{stop}$ and $n < \infty$.*

Note that this definition only accounts for *successful termination*. The notion of observables we consider does not include infinite computation nor those situations in which the agent in the final configuration is not the **stop** agent and yet is unable to make a transition, i.e. the case of *suspended computations*. We denote by $\mathrm{Comp}(A, d)$ the set of all computational paths for A in store d.

Definition 2. *Let $\pi \in \mathrm{Comp}(A, d)$ be a computational path for A in store d $\pi \equiv \langle A, d\rangle = \langle A_0, c_0\rangle \longrightarrow_{p_1} \langle A_1, c_1\rangle \longrightarrow_{p_2} \ldots \longrightarrow_{p_n} \langle A_n, c_n\rangle$. We define the result of π as $res(\pi) = c_n$ and its probability as $prob(\pi) = \prod_{i=1}^{n} p_i$.*

Given a PCCP agent A, the set $\mathcal{R}$ of the results of A is the family of all pairs $\langle c, p\rangle$, where c is the final store corresponding to the least upper bound of the partial constraints accumulated during a computational path, and p is the probability of reaching that result:

$$\mathcal{R}(A, d) = \{\langle c, p\rangle \mid \text{ there exists } \pi \in \mathrm{Comp}(A, d) : c = res(\pi) \text{ and } p = prob(\pi)\}.$$

There might be different computational paths leading to the same result. Thus, we need to "compactify" the results so as to identify all those pairs with the same constraint as a first component. To this purpose we introduce the following equivalence relation on $\mathrm{Comp}(A, d)$.

Definition 3. *Let $\pi, \pi' \in \mathrm{Comp}(A, d)$ be two computational paths for A in store d. We define $\pi \equiv \pi'$ iff $res(\pi) = res(\pi')$. The equivalence class of π is denoted by $[\pi]$.*

The definitions of $res(\pi)$ and $prob(\pi)$ are extended to $\mathrm{Comp}(A)/_{\equiv}$ in the obvious way by $res([\pi]) = res(\pi)$ and $prob([\pi]) = \sum_{\pi' \in [\pi]} prob(\pi')$. Thus, the compactification of the results of a given agent A in store d corresponds to the set $\mathcal{K}(\mathcal{R}(A, d)) = \{\langle res([\pi]), prob([\pi])\rangle \mid [\pi] \in \mathrm{Comp}(A)/_{\equiv}\}$.

We can now define the observables of an agent A with respect to store d as:

$$\mathcal{O}(A, d) = \mathcal{K}(\mathcal{R}(A, d)).$$

The observables $\mathcal{O}(A, d)$ are actually a probability distribution on the constraint system $\mathcal{C}$ and can be seen as a vector in the free vector space associated to $\mathcal{C}$, namely:

$$\mathcal{V}(\mathcal{C}) = \left\{ \sum x_c c \mid x_c \in \mathbb{R},\ c \in \mathcal{C} \right\}.$$

Each linear combination $\sum x_c c$ of constraints represents a vector distribution and its coordinates x_c correspond to the probabilities associated to each constraint.

The notion $\mathcal{O}(A, d)$ can be extended so as to consider the results of executing the agent A starting from a given distribution over constraints instead of a single constraint. The resulting notion corresponds to the vectorial sum of the observables of A in each constraint weighted by the probability associated to that store. By a slight abuse of notation we will denote this extended notion by $\mathcal{O}(A, \boldsymbol{d})$, where $\boldsymbol{d}$ is a store distribution. Formally, this is defined as:

$$\mathcal{O}(A, \boldsymbol{d}) = \sum \{p \cdot \mathcal{O}(A, c) \mid p = \boldsymbol{d}(c) \neq \emptyset\}.$$

3 Confinement

We consider in the following the notion of *identity confinement* [6]. This notion establishes whether it is possible to identify which process is running in a given program. We will look at different types of relatively simple attacks aiming to reveal the identity of some unknown process. Our aim is to characterise the "vulnerability" of a set of agents by applying program analysis techniques. In particular, we will use the Probabilistic Abstract Interpretation technique [10, 11], which we will briefly recall in Section 4.1, to define a suitable collecting semantics for our analysis.

Example 1. In an imperative language, confinement — as formulated for example in [18] — usually refers to a standard (two-level) security model consisting of

high and low level variables. One then considers the (value of the) high variable h as confined if the value of the low level variable l is not "influenced" by the value of the high variable, i.e. if the observed values of l are independent of the values of h.

The following statement illustrates the difference between non-deterministic and probabilistic confinement [17,18]:

$$h := h \bmod 2; (l := h \;{}_{\frac{1}{2}}[]_{\frac{1}{2}}\; (l := 0 \;{}_{\frac{1}{2}}[]_{\frac{1}{2}}\; l := 1))$$

The value of l clearly depends "somehow" on h. However, if we resolve the choice non-deterministically it is impossible to say anything about the value of h by observing the possible values of l. Concretely, we get the following dependencies between h and possible values of l: For $h \bmod 2 = 0$ we have $\{l = 0, l = 1\}$ and for $h \bmod 2 = 1$ we get $\{l = 1, l = 0\}$, i.e. the possible values of l are the same independently from the fact that h is even or odd. In other words, h is non-deterministically confined.

In a probabilistic setting the observed values for l and their probabilities allow us to distinguish cases where h is even from those where h is odd. We have the following situation: For $h \bmod 2 = 0$ we get $\left\{\left\langle \frac{3}{4}, l = 0\right\rangle, \left\langle \frac{1}{4}, l = 1\right\rangle\right\}$, and for $h \bmod 2 = 1$ we have $\left\{\left\langle \frac{1}{4}, l = 0\right\rangle, \left\langle \frac{3}{4}, l = 1\right\rangle\right\}$. Therefore, the probabilities to get $l = 0$ and $l = 1$ reveal if h is even or odd, i.e. h is probabilistically *not* confined.

Example 2. A similar situation to Example 1 can be described in our declarative setting by the following agents:

$$\begin{aligned}
\texttt{highOn} &\equiv \mathbf{ask}(\mathit{true}) \rightarrow \frac{1}{2} : \mathbf{tell}(\texttt{on}) \;[]\; \mathbf{ask}(\mathit{true}) \rightarrow \frac{1}{2} : \texttt{Random} \\
\texttt{highOff} &\equiv \mathbf{ask}(\mathit{true}) \rightarrow \frac{1}{2} : \mathbf{tell}(\texttt{off}) \;[]\; \mathbf{ask}(\mathit{true}) \rightarrow \frac{1}{2} : \texttt{Random} \\
\texttt{Random} &\equiv \mathbf{ask}(\mathit{true}) \rightarrow \frac{1}{2} : \mathbf{tell}(\texttt{on}) \;[]\; \mathbf{ask}(\mathit{true}) \rightarrow \frac{1}{2} : \mathbf{tell}(\texttt{off})
\end{aligned}$$

The constraint system underlying these agents has four elements

$$\mathcal{C} = \{\mathit{true}, \texttt{on}, \texttt{off}, \mathit{false} = \texttt{on} \sqcup \texttt{off}\},$$

with the ordering $\mathit{true} \sqsubseteq \texttt{on} \sqsubseteq \mathit{false}$ and $\mathit{true} \sqsubseteq \texttt{off} \sqsubseteq \mathit{false}$. The constraints `on` and `off` represent the situations in which the low variable $l = 1$ or $l = 0$ respectively. The agent `highOn` corresponds then to the behaviour of the imperative program fragment in case that $h \bmod 2 = 1$ while `highOff` corresponds to the case where $h \bmod 2 = 0$. The auxiliary agent `Random` corresponds to the second choice in the above imperative fragment. The imperative notion of confinement now translates in our framework into a problem of identity confinement: Getting information about h in the previous setting is equivalent to discriminating between `highOn` and `highOff`, i.e. revealing their identity.

3.1 Approximate Confinement

The definition of the notion of *approximate confinement* we give is parametric with respect to a set of admissible spies $\mathcal{S}$ and can be instantiated for different types of attacks. We say that two agents A and B are *approximately confined* with respect to a set of spies $\mathcal{S}$ if there exists an $\varepsilon \geq 0$ such that for all $S \in \mathcal{S}$ the *distance* between the observables of agent A attacked by S and the observables of agent B attacked by S is smaller than ε. The precise definition of what "attacked by S" means obviously depends on the nature of the attack and will be made clear later on. In any case, as we refer to probabilistic observables (i.e. vector distributions) we will use as a measure for this distance between observables the supremum norm $\|\cdot\|_\infty$ formally defined as $\|(x_i)_{i\in I}\| = \|(x_i)_{i\in I}\|_\infty = \sup_{i\in I} |x_i|$, where $(x_i)_{i\in I}$ represents a probability distribution. This norm allows us to identify a single constraint c for which the associated probabilities are maximally different and is therefore particularly appropriate for our purposes.

Admissible Spies. We consider only *passive* spies, i.e. spies which are sensitive to the computational states or environments created by the agents they attack but which are unable to change the execution of the attacked agent. Moreover, our spies are *memoryless* spies, i.e. spies whose execution depends only on the single state or environment they are executed in.

Concretely we therefore consider spies of type $\mathcal{S}_n$ which are PCCP agents of the following form:

$$\mathcal{S}_n = \left\{ []_{i=1}^{n} \mathbf{ask}(c_i) \to q_i : \mathbf{tell}(f_i) \right\},$$

where $f_i \in \mathcal{C}$ are *fresh* constraints, that is constraints which never appear in the execution of the host agents, and $c_i \in \mathcal{C}$ are some guards.

These spies may interact in various ways with the agents in question. We will consider two different models based on an *internal* and an *external* scheduling respectively.

Internal Scheduling: Trojan Horses. The spy $S \in \mathcal{S}_n$ can be considered to be part of the system, i.e. it is treated like a normal PCCP agent which is executed in parallel with the attacked agent A. The behaviour we thus have to analyse is that of the agent: $q : S \;\parallel\; p : A$ with $p + q = 1$. We can then define formally the approximate confinement property for internal attacks as follows:

Definition 4. *Given a set of admissible spies $\mathcal{S}$, $\varepsilon \geq 0$, and fixed scheduling priorities p and $q = 1 - p$, we call two agents A and B ε-confined iff:*

$$\sup_{S\in\mathcal{S}} \|\mathcal{O}(p : A \parallel q : S) - \mathcal{O}(p : B \parallel q : S)\| = \varepsilon.$$

This definition can be generalised to a set of more than two agents. A semantics-based analysis of this property has been presented in [8,7]. It is easy

to verify that if two agents A and B are ε-confined with $\varepsilon = 0$ then they are probabilistically confined in the sense of [22,18,6].

The advantage of this type of attacks is that we can simply use the standard semantics of PCCP to analyse the confinement properties of A. The disadvantage is that the likelihood of S being executed is decreasing exponentially even when the store created by A does not change. This leads to a kind of "logarithmic" nature of the analysis.

Moreover, internally scheduled spies cannot be used to model attacks which are "external" to the system, such as *timing attacks* [13] and *power consumption attacks* [14]. These attacks are based on regular measurements made on the system to find out the time required to perform a given task (e.g. the modular exponentiation in the RSA crypto-system) or to find out some more physical information like the switching energy dissipated by the system when transiting from one state to the other. The system may be vulnerable against a statistical analysis of the information acquired by such measurements, (e.g. attackers may be able to break the RSA and many other crypto-systems).

External Scheduling or Uniform Attack. A suitable model for the above mentioned external attacks is a spy which "observes" the agent A from the "outside" for a finite number of execution steps and is able to interfere uniformly during the given execution time interval. As this implies the definition of a scheduler essentially different from the one implicit in the semantics of PCCP, we can not simulate this behaviour using the parallel construct in PCCP. Therefore, we introduce the notation: $S \rhd_n A$, where we fix a maximal observation time n. During the first n steps of the execution of A there is each time a constant probability $\frac{1}{n}$ that S might be executed in the store currently provided by A. Thus, we call such an attack a "uniform" attack, in contrast to the "geometric" nature of the above described internal attack. The notion of approximate confinement can now be defined as follows:

Definition 5. *Given a set of admissible spies $\mathcal{S}$, we call two agents A and B ε-confined for some $\varepsilon \geq 0$ and a fixed observation time n iff:*

$$\sup_{S \in \mathcal{S}} \|\mathcal{O}(S \rhd_n A) - \mathcal{O}(S \rhd_n B)\| = \varepsilon.$$

Operational Semantics of Uniform Attacks. The operational meaning of $S \rhd_n A$ can be formalised in terms of the SOS transition system for PCCP given in Table 1 provided we extend it with a dummy rule **R0**: $\langle \mathbf{stop}, c\rangle \longrightarrow_1 \langle \mathbf{stop}, c\rangle$.

This allows us to deal with the situation in which the observed agent terminates in a number of steps $s < n$ by extending the computation with a cycle on the last configuration. The following rule expresses the uniform attack of a spy S to an agent A:

$$\frac{\begin{array}{c}\langle S, c_t\rangle \longrightarrow_q^* \langle \mathbf{stop}, c_t \sqcup d\rangle \\ \langle A, c\rangle \longrightarrow_{p_1} \langle A_1, c_1\rangle \longrightarrow_{p_2} \ldots \longrightarrow_{p_n} \langle A_n, c_n\rangle\end{array}}{\langle S \rhd_n A, c\rangle \longrightarrow_{\frac{1}{n}\cdot q\cdot \prod_{i=1}^n p_i} \langle A_n, c_n \sqcup d\rangle} \quad \text{for } 1 \leq t \leq n.$$

The idea is the following: Suppose that A starting in c can make n steps $\langle A, c\rangle \longrightarrow_{p_1} \langle A_1, c_1\rangle \longrightarrow_{p_2} \ldots \longrightarrow_{p_n} \langle A_n, c_n\rangle$ .On the other hand, assume that S executed in some intermediate store c_t with $1 \leq t \leq n$ produces store $c_t \sqcup d$ with (accumulated) probability [1] q. The idea is then to insert the execution of $\langle S, c_t\rangle$ before the execution of $\langle A_{t+1}, c_{t+1}\rangle$. In other words the path:

$$\langle A, c\rangle \longrightarrow_{p_1} \ldots \langle A_t, c_t\rangle \longrightarrow_{p_t} \langle A_{t+1}, c_{t+1}\rangle \ldots \longrightarrow_{p_n} \langle A_n, c_n\rangle$$

becomes the path:

$$\langle S \rhd_n A, c\rangle \longrightarrow_{p_1} \ldots \langle S \rhd A_t, c_t\rangle \longrightarrow_q^* \langle A_t, c_t \sqcup d\rangle \longrightarrow_{p_t} \ldots \longrightarrow_{p_n} \langle A_n, c_n \sqcup d\rangle .$$

The probability accumulated along this path is $q \cdot \prod_{i=1}^{n} p_i$. Since the "intrusion" of S can happen at any of the n transitions, each of the paths has a probability $\frac{1}{n}$. Combining the probabilities and by a slight abuse of notation, we can then represent such a computation as:

$$\langle S \rhd_n A, c\rangle \longrightarrow_{\frac{1}{n} \cdot q \cdot \prod_{i=1}^{n} p_i} \langle A_n, c_n \sqcup d\rangle .$$

4 Analysing Approximate Confinement for Uniform Attacks

In this section we will present an analysis of the approximate confinement property with respect to external attackers. The analysis is based on an abstract semantics which associates to an agent its *average store* in a n-steps computation. This is the average of the intermediate stores computed by the agent in a computation truncated after n steps. We will show that the operational meaning of $S \rhd_n A$ is precisely characterised by the observables of S in the average store of A. Thus, the abstract semantics will allow us to calculate the ϵ quantifying the confinement of a program. The abstract semantics is defined via a collecting semantics constructed as a probabilistic abstract interpretation of the SOS semantics for PCCP in Section 2.1. We first briefly recall the basic definitions of the Probabilistic Abstract Interpretation framework in the next section.

4.1 Probabilistic Abstract Interpretation

Probabilistic Abstract Interpretation (PAI) [10,11] recasts the well-known Abstract Interpretation technique due to Cousot and Cousot [4,1,15]) in terms of a probabilistic semantics which replaces the standard order-theoretic one. Thus, linear spaces replace cpo's as semantical domains, so that abstraction and concretisation functions are now linear maps between the concrete and abstract domains. Moreover, the standard way to relate the abstraction and the concretisation functions via the notion of Galois connection is replaced by the use of the Moore-Penrose pseudo-inverse of a linear operator [3].

[1] There will be in general several such computations by S with all the associated probabilities summing up to 1.

We assume that the vector spaces representing the semantical domains are vector spaces with *inner product*, as is the case for finite dimensional spaces and Hilbert spaces. Under this assumption, we can define the analog of a classical Galois connection (α, γ) in a vector space setting as follows.

As in the classical case, the abstraction and concretisation functions preserve the structure of the underlying domains; this corresponds to the abstraction $\boldsymbol{\alpha}$ and the concretisation $\boldsymbol{\gamma}$ being linear maps.

Furthermore, the classical condition for Galois connections that $\alpha \circ \gamma(d) \sqsubseteq d$ can be seen in a linear setting as the condition that $\boldsymbol{\alpha} \circ \boldsymbol{\gamma}$ is the orthogonal projection in the image of $\boldsymbol{\alpha}$, that is $\boldsymbol{\alpha} \circ \boldsymbol{\gamma} = \pi_\alpha$. In the same way $\gamma \circ \alpha(c) \geq c$ turns into $\boldsymbol{\gamma} \circ \boldsymbol{\alpha} = \pi_\gamma$.

These two conditions define the so-called *Moore-Penrose pseudo-inverse* [3]. The Moore-Penrose pseudo-inverse $\mathbf{T}^\dagger$ of a linear operator $\mathbf{T}$ can be defined for real and complex finite-dimensional vector spaces as well as for infinite-dimensional Hilbert spaces [2]; it is unique and always exists.

A probabilistic abstract interpretation is now simply defined by a pair $(\boldsymbol{\alpha}, \boldsymbol{\gamma})$ of linear maps between vector spaces representing the concrete and abstract domain such that $\boldsymbol{\gamma}$ is the Moore-Penrose pseudo-inverse of $\boldsymbol{\alpha}$, i.e. $\boldsymbol{\gamma} = \boldsymbol{\alpha}^\dagger$, and vice versa.

4.2 Concrete Semantics

In Table 2 we define a relation $\longrightarrow$ which dynamically generates all the computational paths of a given program P step by step. This relation can be seen as a functional description of the transition relation $\longrightarrow_p$ in Table 1: it transforms a triple $\langle C_1, C_2, q\rangle \in \longrightarrow_p$ into a triple $\langle C_2, C_3, q'\rangle \in \longrightarrow_p$ iff $C_1 \longrightarrow_q C_2 \longrightarrow_{q''} C_3$, with $q' = q \cdot q''$, is a sub-tree of the computational tree for P.

The premises and the conclusions of each rule can be seen as distributions over sets of pairs of configurations. Therefore, Table 2 actually defines a linear operator on the vector space $\mathcal{V}(\text{Conf} \times \text{Conf})$,

$$\mathbf{F} : \mathcal{V}(\text{Conf} \times \text{Conf}) \mapsto \mathcal{V}(\text{Conf} \times \text{Conf}).$$

We will use $\mathbf{F}$ as a generator of sequences of vectors in $\mathcal{V}(\text{Conf} \times \text{Conf})$. For an agent A, a store c and a fixed length n, these sequences are constructed by iteratively applying $\mathbf{F}$ starting from an initial transition $\langle\langle \text{Init}, \mathit{true}\rangle, \langle \text{A}, c\rangle\rangle$. We represent this initial transition as a vector $\boldsymbol{t}_0$ in $\mathcal{V}(\text{Conf} \times \text{Conf})$. Thus, the concrete n-steps semantics of A in c is given by the sequence $(\boldsymbol{t}_0, \mathbf{F}(\boldsymbol{t}_0), \mathbf{F}^2(\boldsymbol{t}_0), \ldots, \mathbf{F}^n(\boldsymbol{t}_0))$ and represents the n-steps prefixes of any computational paths for A in store c.

Note that the domain of the operator $\mathbf{F}$ (i.e. the free vector space $\mathcal{V}(\text{Conf} \times \text{Conf})$) may be infinite-dimensional. However, as we consider only *finite* observations, the set of reachable configurations for each agent is always finite. Therefore, in constructing the semantics of an agent, we can effectively consider the restriction of $\mathbf{F}$ to finite-dimensional sub-spaces of $\mathcal{V}(\text{Conf} \times \text{Conf})$. This allows us to directly apply the PAI framework recalled in Section 4.1, as shown in the next section.

Table 2. A Concrete Collecting Semantics for PCCP

R1 $\{\ldots, \langle\langle A', c'\rangle, \langle \mathbf{tell}(c), d\rangle, p'\rangle, \ldots\}$
$\longrightarrow \{\ldots, \langle\langle \mathbf{tell}(c), d\rangle, \langle \mathbf{stop}, c \sqcup d\rangle, 1 \cdot p'\rangle, \ldots\}$

R2 $\{\ldots, \langle\langle A', c'\rangle, \langle\rangle_{i=1}^{n} \mathbf{ask}(c_i) \to p_i : A_i, d\rangle, p'\rangle, \ldots\}$
$\longrightarrow \{\ldots, \langle\langle\rangle_{i=1}^{n} \mathbf{ask}(c_i) \to p_i : A_i, d\rangle \langle A_j, d\rangle, \tilde{p}_j \cdot p'\rangle, \ldots\}$
$j \in [1, n]$ and $d \vdash c_j$

R3 $\{\ldots, \langle\langle A', c'\rangle, \langle\|_{i=1}^{n}\ p_i : A_i, d\rangle, p'\rangle, \ldots\}$
$\longrightarrow \{\ldots, \langle\langle\|_{i=1}^{n}\ p_i : A_i, d\rangle, \langle A'_j, d'\rangle, \tilde{p}_j \cdot p'\rangle, \ldots\}$
$j \in [1, n]$ and $\langle A_j, d\rangle \longrightarrow_{p_j} \langle A'_j, d'\rangle$

R4 $\{\ldots, \langle\langle A', c'\rangle, \langle \exists_x A, d\rangle, p'\rangle, \ldots\}$
$\longrightarrow \{\ldots, \langle\langle \exists_x A, d\rangle, \langle \exists_x A', d \sqcup \exists_x d'\rangle, p \cdot p'\rangle, \ldots\}$
$\langle A, \exists_x d\rangle \longrightarrow_p \langle A', d'\rangle$

R5 $\{\ldots, \langle\langle A', c'\rangle, \langle p(y), d\rangle, p'\rangle, \ldots\}$
$\longrightarrow \{\ldots, \langle\langle p(y), d\rangle, \langle A, d\rangle, 1 \cdot p'\rangle, \ldots\}$
$p(x) :- A \in P$

Example 3. For the agents `highOn` and `highOff` in Example 2 the only transitions involved are enumerated as follows:

	`highOn`	`highOff`
1	$\langle\langle \texttt{Init}, \mathit{true}\rangle, \langle \texttt{highOn}, \mathit{true}\rangle\rangle$	$\langle\langle \texttt{Init}, \mathit{true}\rangle, \langle \texttt{highOff}, \mathit{true}\rangle\rangle$
2	$\langle\langle \texttt{highOn}, \mathit{true}\rangle, \langle \mathbf{stop}, \texttt{on}\rangle\rangle$	$\langle\langle \texttt{highOff}, \mathit{true}\rangle, \langle \mathbf{stop}, \texttt{off}\rangle\rangle$
3	$\langle\langle \texttt{highOn}, \mathit{true}\rangle, \langle \texttt{Random}, \mathit{true}\rangle\rangle$	$\langle\langle \texttt{highOff}, \mathit{true}\rangle, \langle \texttt{Random}, \mathit{true}\rangle\rangle$
4	$\langle\langle \texttt{Random}, \mathit{true}\rangle, \langle \mathbf{stop}, \texttt{on}\rangle\rangle$	$\langle\langle \texttt{Random}, \mathit{true}\rangle, \langle \mathbf{stop}, \texttt{on}\rangle\rangle$
5	$\langle\langle \texttt{Random}, \mathit{true}\rangle, \langle \mathbf{stop}, \texttt{off}\rangle\rangle$	$\langle\langle \texttt{Random}, \mathit{true}\rangle, \langle \mathbf{stop}, \texttt{off}\rangle\rangle$
6	$\langle\langle \mathbf{stop}, \texttt{on}\rangle, \langle \mathbf{stop}, \texttt{on}\rangle\rangle$	$\langle\langle \mathbf{stop}, \texttt{on}\rangle, \langle \mathbf{stop}, \texttt{on}\rangle\rangle$
7	$\langle\langle \mathbf{stop}, \texttt{off}\rangle, \langle \mathbf{stop}, \texttt{off}\rangle\rangle$	$\langle\langle \mathbf{stop}, \texttt{off}\rangle, \langle \mathbf{stop}, \texttt{off}\rangle\rangle$

Therefore, the concrete semantics for these agents can be constructed by using the restrictions of the operator $\mathbf{F}$ represented by the following matrices:

$$\mathbf{F}_{\texttt{highOn}} = \begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{F}_{\texttt{highOff}} = \begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

4.3 Abstract Semantics

In order to define our analysis we will consider a more abstract semantics which we will construct as a probabilistic abstract interpretation of the operator $\mathbf{F}$ introduced in Section 4.2.

We define such a collecting semantics for PCCP via the abstract linear operator $\mathbf{G}$ induced by the probabilistic abstraction $\boldsymbol{\alpha} : \mathcal{V}(\mathrm{Conf} \times \mathrm{Conf}) \mapsto \mathcal{V}(\mathrm{Conf})$ defined as:

$$\boldsymbol{\alpha}(\{\ldots, \langle\langle A_1, c_1\rangle, \langle A_2, c_2\rangle, p\rangle, \ldots\}) = (\{\ldots, \langle\langle A_2, c_2\rangle, p\rangle, \ldots\}).$$

This abstraction only records the last configuration of a transition by transforming a distribution on transitions into a distribution on single configurations. Intuitively, this means that we only record the probability of reaching a given configuration, while the information on the configurations from which it can be reached is lost.

The operator $\mathbf{G} : \mathcal{V}(\mathrm{Conf}) \mapsto \mathcal{V}(\mathrm{Conf})$ is defined as

$$\mathbf{G} = \boldsymbol{\alpha} \cdot \mathbf{F} \cdot \boldsymbol{\gamma},$$

where $\boldsymbol{\gamma}$ is the Moore-Penrose pseudo-inverse of the abstraction $\boldsymbol{\alpha}$ above.

Proposition 1. *The operator* $\mathbf{G} : \mathcal{V}(\mathrm{Conf}) \mapsto \mathcal{V}(\mathrm{Conf})$ *defined as* $\boldsymbol{\alpha} \cdot \mathbf{F} \cdot \boldsymbol{\gamma}$ *transforms distributions over configurations into distributions over configurations according to the rules in Table 3.*

Note that because of nondeterminism the rules in Table 3 actually transform distributions over multi-sets of configurations. Instead the linear operator $\mathbf{G}$ operates on the free vector space over Conf and therefore on distributions over sets of configurations. Therefore the correspondence in Proposition 1 holds up to an appropriate reduction of distributions on multi-sets to distributions on sets. One such reduction would transform a distribution of the form $\{\langle a, 1/2\rangle, \langle b, 1/4\rangle, \langle a, 1/4\rangle\}$ into the distribution $\{\langle a, 3/4\rangle, \langle b, 1/4\rangle\}$.

Analogously to the concrete semantics, the collecting semantics of an agent is given for a fixed length n, by the sequence $(\mathbf{G}^i(\boldsymbol{\alpha}\boldsymbol{t}_0))_{i=0}^n$.

The semantics $\mathbf{G}$ allows us to construct for a program P an abstract representation of its computational tree given by the sequence of the sets of nodes at each level of the tree. The length n of the sequence represents the maximal depth of the tree we need to construct for our observation. Clearly this abstract representation does not allow us to re-construct exactly all the computational paths of P.

Example 4. Consider again Example 3. To obtain the abstract semantics for the agents `highOn` and `highOff`, we first enumerate all those configurations in Conf which are reachable by `highOn` and `highOff` respectively:

$$\langle \texttt{highOn}, \mathit{true}\rangle, \langle \texttt{Random}, \mathit{true}\rangle, \langle \mathbf{stop}, \texttt{on}\rangle, \langle \mathbf{stop}, \texttt{off}\rangle$$
$$\langle \texttt{highOff}, \mathit{true}\rangle, \langle \texttt{Random}, \mathit{true}\rangle, \langle \mathbf{stop}, \texttt{on}\rangle, \langle \mathbf{stop}, \texttt{off}\rangle.$$

Table 3. An Abstract Collecting Semantics for PCCP

R1	$\{\ldots, \langle\langle \mathbf{tell}(c), d\rangle, p\rangle, \ldots\} \longrightarrow \{\ldots, \langle\langle \mathbf{stop}, c \sqcup d\rangle, p\rangle, \ldots\}$
R2	$\{\ldots, \langle\langle \bigsqcup_{i=1}^{n} \mathbf{ask}(c_i) \to p_i : A_i, d\rangle, q\rangle, \ldots\} \longrightarrow \{\ldots, \langle\langle A_j, d\rangle, q \cdot \tilde{p}_j\rangle, \ldots\}$ $j \in [1, n]$ and $d \vdash c_j$
R3	$\{\ldots, \langle\langle \|_{i=1}^{n}\ p_i : A_i, d\rangle, q\rangle, \ldots\} \longrightarrow \{\ldots, \langle\langle A'_j, d'\rangle, q \cdot \tilde{p}_j\rangle, \ldots\}$ $j \in [1, n]$ and $\langle A_j, d\rangle \longrightarrow_{p_j} \langle A'_j, d'\rangle$
R4	$\{\ldots, \langle\langle \exists_x A, d\rangle, q\rangle, \ldots\} \longrightarrow \{\ldots, \langle\langle \exists_x A', d \sqcup \exists_x d'\rangle, q \cdot p\rangle, \ldots\}$ $\langle A, \exists_x d\rangle \longrightarrow_p \langle A', d'\rangle$
R5	$\{\ldots, \langle\langle p(y), d\rangle, q\rangle, \ldots\} \longrightarrow \{\ldots, \langle\langle A, d\rangle, q\rangle, \ldots\}$ $p(x) : -A \in P$

Next we calculate the abstractions $\boldsymbol{\alpha}_{\texttt{highOn}}$ and $\boldsymbol{\alpha}_{\texttt{highOff}}$, which map transitions into their final configurations — e.g. $\langle\langle \texttt{highOn}, \mathit{true}\rangle, \langle \texttt{Random}, \mathit{true}\rangle\rangle$ is mapped into $\langle \texttt{Random}, \mathit{true}\rangle$ — and their corresponding Moore-Penrose Pseudo-Inverses $\boldsymbol{\gamma}_{\texttt{highOn}}$ and $\boldsymbol{\gamma}_{\texttt{highOff}}$. These maps are represented by the matrices in Table 4. With this we can construct the induced semantics $\mathbf{G}_{\texttt{highOn}} = \boldsymbol{\alpha}_{\texttt{highOn}}\mathbf{F}_{\texttt{highOn}}\boldsymbol{\gamma}_{\texttt{highOn}}$ and $\mathbf{G}_{\texttt{highOff}} = \boldsymbol{\alpha}_{\texttt{highOff}}\mathbf{F}_{\texttt{highOff}}\boldsymbol{\gamma}_{\texttt{highOff}}$:

$$\mathbf{G}_{\texttt{highOn}} \begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{G}_{\texttt{highOff}} = \begin{pmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

4.4 Analysis: The Average Store

In order to find out whether two agents A and B are approximately confined with respect to an external attack we have to calculate the observables of $S \rhd_s A$ and $S \rhd_s B$ and evaluate their distance. The semantics for $S \rhd_s A$ as defined in Section 3.1 can be equivalently described by means of the collecting semantics introduced in the previous section. In particular, by a further abstraction we can construct the average store out of $\mathbf{G}(A)$ and $\mathbf{G}(B)$ as follows. We define the linear operator $\boldsymbol{\beta} : \mathcal{V}(\mathrm{Conf}) \mapsto \mathcal{V}(\mathcal{C})$,which transforms distributions over configurations into distributions over stores. Such an operator simply extracts the constraint from a configuration. We then calculate the *average store* of A (B) in n steps by summing up the distributions representing the stores reached by A (B) after t steps, for all $t \in [1, n]$ weighted by $\frac{1}{n}$. Formally, the average store semantics of an agent A is parametric in n and is defined as:

$$[\![A]\!]_n = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{\beta}(\mathbf{G}^i \{\langle A, \mathit{true}\rangle\}).$$

Table 4. Matrix Representation of Operators.

$$\boldsymbol{\alpha}_{\texttt{highOn}} = \begin{pmatrix} 1&0&0&0\\ 0&0&1&0\\ 0&1&0&0\\ 0&0&1&0\\ 0&0&0&1\\ 0&0&1&0\\ 0&0&0&1 \end{pmatrix} \quad \boldsymbol{\gamma}_{\texttt{highOn}} = \begin{pmatrix} 1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0\\ 0&\frac{1}{3}&0&\frac{1}{3}&0&\frac{1}{3}&0\\ 0&0&0&0&\frac{1}{2}&0&\frac{1}{2} \end{pmatrix}$$

$$\boldsymbol{\beta} = \begin{pmatrix} 1&0&0&0\\ 1&0&0&0\\ 0&1&0&0\\ 0&0&1&0 \end{pmatrix}$$

$$\boldsymbol{\alpha}_{\texttt{highOff}} = \begin{pmatrix} 1&0&0&0\\ 0&0&0&1\\ 0&1&0&0\\ 0&0&1&0\\ 0&0&0&1\\ 0&0&1&0\\ 0&0&0&1 \end{pmatrix} \quad \boldsymbol{\gamma}_{\texttt{highOff}} = \begin{pmatrix} 1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0\\ 0&0&0&\frac{1}{2}&0&\frac{1}{2}&0\\ 0&\frac{1}{3}&0&0&\frac{1}{3}&0&\frac{1}{3} \end{pmatrix}$$

Proposition 2. *For an agent A and a spy S we have:*

$$\mathcal{O}(S \rhd_n A) = \mathcal{O}(S, [\![A]\!]_n).$$

That is: The observables of an n steps attack on A by S can be obtained by simply executing S in the (n-step) average store of A. Therefore, the average store of an agent determines their ε-confinement.

Example 5. The abstraction $\boldsymbol{\beta}$ for the agents `highOn` and `highOff` is represented by the matrix in Table 4.

Using the generator $\mathbf{G}$ from Example 4 for the abstract semantics we can now (re)construct the average stores, using the following operators:

$$\frac{1}{n}\left(\sum_{i=0}^{n-1} \boldsymbol{\beta} \cdot \mathbf{G}^i_{\texttt{highOn}}\right) \text{ and } \frac{1}{n}\left(\sum_{i=0}^{n-1} \boldsymbol{\beta} \cdot \mathbf{G}^i_{\texttt{highOff}}\right).$$

By applying these two operators to the initial configurations: $\boldsymbol{t}_{\texttt{highOn}} = \{\langle\langle \texttt{highOn}, \mathit{true}\rangle, 1\rangle\} = (1,0,0,0,0)$ and $\boldsymbol{t}_{\texttt{highOff}} = \{\langle\langle \texttt{highOff}, \mathit{true}\rangle, 1\rangle\} = (0,1,0,0,0)$, we obtain the average stores

$$[\![\texttt{highOn}]\!]_3 = \left\{\left\langle \mathit{true}, \frac{1}{2}\right\rangle, \left\langle \texttt{on}, \frac{5}{12}\right\rangle, \left\langle \texttt{off}, \frac{1}{12}\right\rangle\right\}$$

$$[\![\texttt{highOff}]\!]_3 = \left\{\left\langle \mathit{true}, \frac{1}{2}\right\rangle, \left\langle \texttt{on}, \frac{1}{12}\right\rangle, \left\langle \texttt{off}, \frac{5}{12}\right\rangle\right\}$$

This shows that `highOn` and `highOff` can be distinguished by an external spy observing both agents for three steps. One such external spy is given by:

$$S \equiv \mathbf{ask}(\texttt{on}) \to \frac{1}{3} : \mathbf{tell}(e) \;[]\; \mathbf{ask}(\texttt{off}) \to \frac{2}{3} : \mathbf{tell}(f).$$

For a three steps external attack of S on `highOn` and `highOff` we get the following observables:

$$\mathcal{O}(S \rhd_3 \texttt{highOn}) = \left\{ \left\langle \mathit{true}, \frac{1}{2} \right\rangle, \left\langle \texttt{on} \sqcup e, \frac{5}{12} \right\rangle, \left\langle \texttt{off} \sqcup f, \frac{1}{12} \right\rangle \right\}$$
$$\mathcal{O}(S \rhd_3 \texttt{highOff}) = \left\{ \left\langle \mathit{true}, \frac{1}{2} \right\rangle, \left\langle \texttt{on} \sqcup e, \frac{1}{12} \right\rangle, \left\langle \texttt{off} \sqcup f, \frac{5}{12} \right\rangle \right\}$$

i.e. with respect to three steps external attacks by this particular spy S the two agents `highOn` and `highOff` are $\frac{1}{3}$-confined.

Example 6. In this example we show how timing attacks [13] can be modelled as external attacks. For any number $m \in \mathbb{N}$, consider the following PCCP agents:

$$\begin{aligned} \texttt{Countdown}(m) :- \ & \mathbf{ask}(m = 0) \rightarrow 1 : \mathbf{tell}(\texttt{done}) \\ [] \ & \mathbf{ask}(m \neq 0) \rightarrow 1 : \texttt{Countdown}(m-1). \end{aligned}$$

As long as such an agent $\texttt{Countdown}(m)$ is just "counting down", i.e. $m \neq 0$, the store will be unchanged. Only when the count-down is finished a marker `done` is placed into the store. For all $m \in \mathbb{N}$ the agent $\texttt{Countdown}(m)$ therefore has the same input/output observables:

$$\mathcal{O}(\texttt{Countdown}(m), \mathit{true}) = \{\langle \texttt{done}, 1 \rangle\}.$$

Consider two agents $\texttt{Countdown}(m_1)$ and $\texttt{Countdown}(m_2)$, with $m_1 < m_2$. As long as both agents are counting down, no spy will observe any difference, i.e. an attack $S \rhd_n \texttt{Countdown}(m)$ is pointless for $n \leq m_1$. After the first agent terminates the difference between the two agent may be observed by a spy like:

$$S \equiv \mathbf{ask}(\mathit{true}) \rightarrow \frac{1}{2} : \mathbf{tell}(a) \ [] \ \mathbf{ask}(\texttt{done}) \rightarrow \frac{1}{2} : \mathbf{tell}(b),$$

where a and b are fresh constraints, once it is scheduled for a time slot $n > m_1$. After also the second agent terminates, the spy will again see no difference between the two agents.

The most effective spy thus has to observe two agents $\texttt{Countdown}(m_1)$ and $\texttt{Countdown}(m_2)$ for exactly m_2 steps. In this case the average stores

$$\begin{aligned} [\![\texttt{Countdown}(m_1)]\!]_{m_2} &= \{\langle \mathit{true}, \frac{m_1}{m_2} \rangle, \langle \texttt{done}, \frac{m_2 - m_1}{m_2} \rangle\} \\ [\![\texttt{Countdown}(m_2)]\!]_{m_2} &= \{\langle \mathit{true}, 1 \rangle\} \end{aligned}$$

are maximally different. To calculate this difference we first construct the observables for S spying for m_2 steps on $\texttt{Countdown}(m_1)$ and $\texttt{Countdown}(m_2)$:

$$\mathcal{O}(S \rhd_{m_1} \texttt{Countdown}(m_1)) = \mathcal{O}(S, [\![\texttt{Countdown}(m_1)]\!]_{m_2})$$

$$
\begin{aligned}
&= \mathcal{O}(S, \{\langle \mathit{true}, \frac{m_1}{m_2}\rangle, \langle \mathtt{done}, \frac{m_2 - m_1}{m_2}\rangle\}) \\
&= \{\langle a, \frac{m_1}{m_2} + \frac{1}{2}(\frac{m_2 - m_1}{m_2})\rangle, \langle b, \frac{1}{2}(\frac{m_2 - m_1}{m_2})\rangle\} \\
\mathcal{O}(S \rhd_{m_2} \mathtt{Countdown}(m_2)) &= \mathcal{O}(S, [\![\mathtt{Countdown}(m_1)]\!]_{m_2}) \\
&= \mathcal{O}(S, \{\langle \mathit{true}, 1\rangle\}) \\
&= \{\langle a, 1\rangle\}.
\end{aligned}
$$

Thus we have:

$$
\|\mathcal{O}(S \rhd_{m_1} \mathtt{Countdown}(m_1)) - \mathcal{O}(S \rhd_{m_2} \mathtt{Countdown}(m_2))\| = \frac{m_2 - m_1}{2m_2}.
$$

5 Conclusions

We have defined a notion of approximate confinement and shown how it can be used to analyse the robustness of systems in the presence of externally scheduled spies. This analysis is based on a state-based collecting semantics constructed as a probabilistic abstract interpretation of the concrete SOS semantics. For large choices of n or for the analysis of languages that do not share the nice monotonicity properties of PCCP, it would be necessary to construct a static analysis. This might be constructed as probabilistic abstract interpretation of the state-based collecting semantics or in a more ad hoc way — this remains a topic for future investigation.

References

1. S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis-Horwood, Chichester, England, 1987.
2. F.J. Beutler. The operator theory of the pseudo-inverse. *Journal of Mathematical Analysis and Applications*, 10:451–470,471–493, 1965.
3. S.L. Campbell and D. Meyer. *Generalized Inverse of Linear Transformations.* Constable and Company, London, 1979.
4. P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2-3):103–180, July 1992.
5. F.S. de Boer, A. Di Pierro, and C. Palamidessi. Nondeterminism and Infinite Computations in Constraint Programming. *Theoretical Computer Science*, 151(1):37–78, 1995.
6. A. Di Pierro, C. Hankin, and H. Wiklicky. Probabilistic confinement in a declarative framework. In *Declarative Programming – Selected Papers from AGP 2000 – La Havana, Cuba*, volume 48 of *Electronic Notes in Theoretical Computer Science*, pages 1–23. Elsevier, 2001.
7. A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In Iliano Cervesato, editor, *CSFW'02 — 15th IEEE Computer Security Foundation Workshop*, pages 3–17, Cape Breton, Canada, 24–26 June 2002. IEEE Computer Society Press.

8. A. Di Pierro, C. Hankin, and H. Wiklicky. On approximate non-interference. In P. Syverson and J. Guttman, editors, *Proceedings of WITS'02 – Workshop on Issues in the Theory of Security, 14–15 January, Portland*, January 2002. `http://www.dsi.unive.it/IFIPWG1_7/WITS2002`.
9. A. Di Pierro and H. Wiklicky. An operational semantics for Probabilistic Concurrent Constraint Programming. In P. Iyer, Y. Choo, and D. Schmidt, editors, *ICCL'98 – International Conference on Computer Languages*, pages 174–183. IEEE Computer Society Press, 1998.
10. A. Di Pierro and H. Wiklicky. Concurrent Constraint Programming: Towards Probabilistic Abstract Interpretation. In M. Gabbrielli and F. Pfenning, editors, *Proceedings of PPDP'00 – Principles and Practice of Declarative Programming*, pages 127–138, Montréal, Canada, September 2000. ACM SIGPLAN, Association of Computing Machinery.
11. A. Di Pierro and H. Wiklicky. Measuring the precision of abstract interpretations. In *Proceedings of LOPSTR'00 – 10th International Workshop on Logic-Based Program Synthesis and Transformation, London, UK*, volume 2042 of *Lecture Notes in Computer Science*, pages 147–164, Berlin – New York, 2001. Springer Verlag.
12. J. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.
13. P.C. Kocher. Cryptanalysis of Diffie-Hellman, RSA, DSS, and other crypto-systems using timing attacks. In D. Coppersmith, editor, *Advances in Cryptology, CRYPTO '95: 15th Annual International Cryptology Conference, Santa Barbara, California, USA, August 27–31, 1995*, volume 963 of *Lecture Notes in Computer Science*, pages 171–183, Berlin — Heidelberg — London, 1995. Springer-Verlag.
14. P.C. Kocher, J.M. Jaffe, and B Jun. Differential power analysis. In *Proc. 19th International Advances in Cryptology Conference – CRYPTO '99*, pages 388–397, 1999.
15. F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, Berlin – Heidelberg, 1999.
16. P.Y.A. Ryan, J. McLean, J. Millen, and V. Gilgor. Non-interference, who needs it? In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 237–238, Cape Breton, Nova Scotia, Canada, June 2001. IEEE.
17. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *ESOP'99*, number 1576 in Lecture Notes in Computer Science, pages 40–58. Springer Verlag, 1999.
18. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200–214, 2000.
19. V.A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Symposium on Principles of Programming Languages (POPL)*, pages 333–353. ACM, 1991.
20. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Symposium on Principles of Programming Languages (POPL'98)*, pages 355–364, San Diego, California, 1998. ACM.
21. G. Smith and D. Volpano. Verifying secrets and relative secrecy. In *Symposium on Principles of Programming Languages (POPL'00)*, pages 368–276, Boston, Massachusetts, 2000. ACM.
22. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop (CSFW '98)*, pages 34–43, Washington - Brussels - Tokyo, June 1998. IEEE.

An Improved Constraint-Based System for the Verification of Security Protocols

Ricardo Corin[1] and Sandro Etalle[1,2]

[1] Faculty of Computer Science,
University of Twente, P.O.Box 217, 7500AE Enschede, The Netherlands
[2] CWI, Center for Mathematics and Computer Science Amsterdam
{corin,etalle}@cs.utwente.nl

Abstract. We propose a constraint-based system for the verification of security protocols that improves upon the one developed by Millen and Shmatikov [30]. Our system features (1) a significantly more efficient implementation, (2) a monotonic behavior, which also allows to detect flaws associated to partial runs and (3) a more expressive syntax, in which a principal may also perform explicit checks. In this paper we also show why these improvements yield a more effective and practical system.

1 Introduction

Cryptographic protocols are the essential means for the exchange of confidential information and for authentication. Their correctness and robustness are crucial for guaranteeing that a hostile intruder can not get hold of secret information (e.g. a private key) or to force unjust authentication. Unfortunately, the design of cryptographic protocols appears to be rather error-prone. A great deal of published protocols has later been shown to contain errors prejudicing their safety. This stimulated research on formal verification of security protocols see e.g. [19,10,28,31,27,35,36,32,17,18]. In this setting several approaches are based on Dolev and Yao's [19], where it is proposed to test a protocol explicitly against a hostile intruder who has complete control over the network, and who can intercept and forge messages. By an exhaustive search, one can establish whether the protocol is flawed or not [31,29,12,23]. A crucial aspect in this approach is try to limit the search space explosion that occurs when modeling the intruder's behavior.

In this respect, an elegant and effective solution was recently proposed by Millen and Shmatikov [30] (we will refer to their system as MS in the future). Security properties, such as secrecy and authentication, can be characterized as reachability problems; in MS it is shown how to convert a reachability problem into a constraint solving problem. They use the strand space model (originally developed by Thayer, Herzog and Guttman [21]) for honest processes, and a term set closure characterization for the attacker. One limitation of this approach is that the number of sessions of the protocol must be bounded. This is the price one must pay for an automatic full decision procedure.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 326–341, 2002.
© Springer-Verlag Berlin Heidelberg 2002

At the University of Twente, we have extensively used MS for the analysis of security and non-repudiation protocols. The system we present here is an improvement and extension of MS, dictated by our practical experience. In particular, this system improves on MS thanks to the following features:

- **A *monotonic* behavior.** As part of the specification, a system scenario must be declared, stating which protocol instances should be present in the system to be checked. This scenario, called *semibundle*, will be defined more precisely in section 2. With *monotonocity* we indicate the fact that, the system analyses not only the traces for the original semibundle but also possible solutions for *smaller* semibundles. Dually, flaws exhibited by a semibundle are exhibited by larger semibundles as well. This is important in practice. For instance, it allows to easily detect flaws associated to incomplete runs, like the one shown by Lowe [26] on the Woo and Lam mutual authentication protocol [40].
- **More expressiveness.** In particular, it allows the principals to perform explicit checks. Security protocols may perform tests at some stage of their execution. To model this in a natural yet accurate way, it is necessary to extend the definition of strands. We add a new special operation called the *check* operation. This action can be performed by any honest principal when she wants to perform a test for deciding whether to continue or not her protocol execution. This newly added operation improves the expressiveness of the system and allows the user to verify a broader class of protocols.
- **A more efficient semantics.** We modified the semantics of the system to make it simpler and more intuitive. As it turns out, this change leads to a significantly more efficient execution.

This paper is organized as follows. Section 2 contains the preliminary definitions and the description of the new semantics. In Section 3 we show how *explicit checks* work, and why they are necessary. In Section 4 we show that the system exhibits a monotonic behavior, and why this has practical advantages. Furthermore, in Section 5 we report some benchmarks. Conclusions are drawn in Section 6 and finally we conclude in Section 7 mentioning some related work.

2 Constraint-Based Protocol Analysis

2.1 Preliminaries

The reader is assumed to be familiar with the terminology and the basic results of the semantics of logic programs [4,24]. In particular, we assume familiarity with the concept of (most general) unifier. Moreover, we use Prolog notation for distinguishing variables (that start with an uppercase letter, or an underscore) from non-variable terms. The only exception to this rule is that we follow the standard security notation in which names of principals are denoted by uppercase letters (A, B, I).

Parametric strands. We now introduce in an informal way the basic notions used in the sequel. For a more formal approach we refer to [30,21,39]. Given a term t, $+t$ and $-t$ are *events*, in particular $+t$ is a *send* event, and $-t$ is a *receive* event; in both cases, t is the contents of the message. Messages are built according to the free algebra generated by the following operators (we show in parenthesis the notation adopted in the implementation):

- $[t1, t2]$ (`[t1,t2]`) pairing.
- $pk(P)$ (`pk(P)`) public key of P.
- $h(t)$ (`sha(t)`) the hash of t.
- $[t]_k^{\leftrightarrow}$ (`t+k`) term t encrypted with k using a symmetric key algorithm.
- $[t]_k^{\rightarrow}$ (`t*k`) term t encrypted with k using a public key algorithm.
- $sig_k(t)$ (`t/k`) public key signature of t that is validated using key k. We assume that private keys of a key pair are never leaked, thus the attacker can only construct its own signatures.
- ε (`e`). Constant representing the identity of the attacker.

A sequence of events is called a (parametrized, as it might contain variables) *strand*. A strand can represent thus the role of a principal in a protocol, and is mostly used so. For instance, the initiator role of the Needham-Schroeder-Lowe public-key protocol handshake (NSL) [25] is specified as:

$$Init(A, B, N_A, N_B) = +[A, N_A]_{pk(B)}^{\rightarrow} - [N_A, N_B, B]_{pk(A)}^{\rightarrow} + [N_B]_{pk(B)}^{\rightarrow}$$

The responder role $Resp(A, B, N_A, N_B)$ is the same except that $+$ and $-$ are interchanged.

A *semibundle* is a (finite) set of parametrized strands. Thus, for example,

$$\{Init(a, B, n_a, N_B\}, Resp(A, b, N_A, n_b)\}$$

is a semibundle. Notice that we instantiated A with a and N_A with n_a in $Init$, as well as B with b and N_B with n_b in $Resp$. In the first case, this specifies that we want the honest principal a to take the initiator role, and to introduce the fresh nonce[1] n_a (and similarly for the $Resp$ role). The communication model is that of Dolev and Yao, in which the (malicious) intruder has complete control over the network. Actually the intruder is identified with the network itself: when a principal sends a message, it actually passes it to the intruder; on the other hand, when a principal needs to receive a message, the system checks if the intruder is able to generate a message of the appropriate format.

2.2 A New Semantics

To explain the semantics of the system, we employ a more intuitive (and algorithmic) approach than MS and define the following notion of a (partial) *run*. Given a semibundle S and an intruder's initial knowledge K, a (partial) *run* is a reiterated sequence of the following steps:

[1] A nonce is a randomly generated value, usually employed to defeat replay attacks.

1. a non-empty strand in S is chosen nondeterministically, and its first event e is removed from it,
2. if the event is a send, $e = +t$, then t is added to the knowledge of the intruder, $K := K \cup \{t\}$
3. if the event is a receive, $e = -t$, then it is checked if the attacker ε can generate (using the knowledge K) an instance t' of t. In that case, if $\theta = mgu(t, t')$ then the remaining semibundle S and the intruders knowledge are instantiated by θ (e.g., $S := S\theta$, $K := K\theta$), and the computation proceeds from (1). If ε cannot generate such a message, then the run stops.

To avoid an infinitely branching system, we have to pay attention at step (3), in fact the intruder might be able to generate an infinite number of instances of t. To solve this problem, the constraint solving system works in the following *lazy* fashion: t' is constructed incrementally (by incremental instantiation). As soon as t' becomes an instance of t then the construction process is suspended, (the mgu θ is applied to K and S automatically) and the system proceeds with step (1). The suspended construction process is later resumed in the case that some subsequent action (e.g., a check) instantiates t in such a way that t' is not any longer an instance of it. A resumed construction process yields either to a new (more instantiated) mgu θ or to failure.

The knowledge of the intruder contains initially only the names of the agents involved in the protocol. Clearly, a run contains always a finite number of steps. In MS it is demonstrated that the process of checking whether ε can generate a message t always terminates. Therefore one can generate all runs in a finite time and full nondeterminism is achieved by means of backtracking. We say that the run is *complete* if after the last step the semibundle contains only empty strands, otherwise we call it a *partial run*. (In the notation of MS, a complete run is called a *bundle*).

While we adopted this semantics basically for clarity reasons, it turned out to lead to a significantly faster implementation than MS. This will be explained in Section 5.

Finally, we need the following order relationship $\preceq$.

Definition 1. *Let s and s' be strands and S, S' be semibundles. We say that*

- *$s \preceq s'$, if s is a prefix of s', and that*
- *$S \preceq S'$, If there exists a variable renaming θ for which, for each $s \in S\theta$ there exists a distinct $s' \in S'$ such that $s \preceq s'$.*

3 Explicit Checks

Our first extension consists of the addition of a new event to the strand space, the *check event*. To motivate its necessity, consider the following protocol (inspired by Zhou-Gollman [41]):

Message 1. $A \to B : [L, C]$
Message 2. $B \to A : sig_{pk(B)}(C)$
Message 3. $A \to B : [sig_{pk(A)}(K), K]$
Message 4. $B \to A : sig_{pk(B)}([L, C, K])$

Here, if K is a symmetric key and M a message, $C = [M]_K^{\leftrightarrow}$ is obtained by symmetric encryption of M by K, and finally $L = h(M)$ is obtained by hashing M. In message 1, A starts the session by sending to B the hash L and a ciphered message C. B replies with his signature of C. In message 3, A sends the signed symmetric key K to B (we need to send K also in plain since our signature operator is not invertible). Now, *and not before*, B can decrypt C and check that $L = h(M)$. If the check does not succeed then B stops the session, otherwise he continues by sending message 4.

A Vulnerability. The above protocol presents the following vulnerability: the attacker can impersonate a honest principal (A, the initiator), and convince B that he is communicating with A. The attack comes to light by instantiating the protocol twice (sessions α and β) as follows:

Message $\alpha.1$. $I_A \to B : [h(M), [M]_K^{\leftrightarrow}]$
Message $\alpha.2$. $B \to I_A : sig_{pk(B)}([M]_K^{\leftrightarrow})$
Message $\beta.1$. $I \to A : [*, K]$
Message $\beta.2$. $A \to I : sig_{pk(A)}(K)$
Message $\alpha.3$. $I_A \to B : [sig_{pk(A)}(K), K]$
Message $\alpha.4$. $B \to I_A : sig_{pk(B)}([h(M), [M]_K^{\leftrightarrow}, K])$

First the intruder starts to impersonate A in message $\alpha.1$ (I_A is the standard notation for indicating the intruder I impersonating the honest principal A). He sends $L = h(M)$ and $C = [M]_K^{\leftrightarrow}$, for some M and K. B then replies signing of $[M]_K^{\leftrightarrow}$. Now the intruder must provide the message $sig_{pk(A)}(K)$. For this, he initiates a new session β with A, and composes a new message with anything as L, represented as $*$ (it does not matter what value it takes) but with K as C (message $\beta.1$). Then A replies by signing K, and this is exactly what the intruder needs (message $\beta.2$). He just creates a new message concatenating this message and K, and sends it to B (message $\alpha.3$). B receives the message, performs successfully the check and finishes his session by sending the last message $\alpha.4$. Note also that the β session is partial.

3.1 Extended Strands

In order to implement the protocol correctly, we need to introduce a new event to the strand model, the *check*. A check event has the syntax *check*$(t_1 = t_2)$, where t_1 and t_2 are terms of the free algebra defined in Section 2. We call *extended* strand (resp. bundles) a parametric strand (bundle) in which check events are allowed. The definition of a *run* is extended in the straightforward way to include check: when a check event is selected, if t_1 and t_2 can be reduced to the same (most general) term t, so that $t_1\theta = t_2\theta = t$, then the remaining semibundle and

the intruder's knowledge are instantiated by θ, and the run proceeds, otherwise the run stops (in this latter case, we have a partial run).

Using extended strands, the above protocol can be specified in Prolog as follows (here we only show the specification of the responder role B, which is sufficient for finding the attack):

```
responder(A, B, L, C, K, M, [recv([L, C]),
                             send(C/pk(B)),
                             recv([K/pk(A),K]),
                             check(C=M+K),
                             check(L=sha(M)),
                             send([L, [C, K]]/pk(B)]).
```

To find the above attack we need to have at least two parallel sessions, actually we need a semibundle with at least two responder roles. In Prolog, the semibundle specification is the following.

```
semibundle([Sa,Sb]):- responder(a,b, L, C1, k, m, Sa),
                      responder(A,B, L, C2, k, m, Sb).
```

Now, what we have to check is whether there exists a run in which the first strand can complete its role. The output we obtained from the system corresponds with the attack described previously:

```
trace:
[responder(a, b, sha(m), m + k, k, m), recv([sha(m), m + k])]
[responder(a, b, sha(m), m + k, k, m), send((m + k) / pk(b))]
[responder(e, a, sha(m), k, k, m), recv([sha(m), k])]
[responder(e, a, sha(m), k, k, m), send(k / pk(a))]
[responder(a, b, sha(m), m + k, k, m), recv([k / pk(a), k])]
[responder(e, a, sha(m), k, k, m), recv([k / pk(e), k])]
[responder(a, b, sha(m), m + k, k, m), check(m + k = m + k)]
[responder(a, b, sha(m), m + k, k, m), check(sha(m) = sha(m))]
[responder(a, b, sha(m), m + k, k, m), send([k, [m + k, sha(m)]] / pk(b))]
reached state:
[[responder(a, b, sha(m), m + k, k, m), []], [responder(e, a, sha(m), k, k,
m), [check(k = m + k), check(sha(m) = sha(m)), send([k, [k, sha(m)]] /
pk(a))]]]
```

The trace reported by the system is a list of pairs of the form `[role(X), event(M)]`; this should be read as "*role* X performs *event* with message M". Exploring the output, we can notice the instantiations of A and B to ε and a respectively given by the system.

3.2 Pattern Matching Approach

We now go back to the original system MS to show that, in that context, the above attack cannot be found in a natural way. Actually, there are two reasons why this is so: the first one is because MS does not allow explicit checks, and the second (also important) reason is that in MS only complete runs are considered, while the above attack comes from a *partial run* (this problem is addressed in Section 4).

Without having explicit checks, the natural way of performing the check after message 3 is by directly substituting in the first line L with $h(M)$ and C with $[M]_K^{\leftrightarrow}$ and modify the rest of the protocol accordingly.

Message 1. $A \to B : [h(M), [M]_K^{\leftrightarrow}]$
Message 2. $B \to A : sig_{pk(B)}([M]_K^{\leftrightarrow})$
Message 3. $A \to B : [sig_{pk(A)}(K), K]$
Message 4. $B \to A : sig_{pk(B)}([h(M), [M]_K^{\leftrightarrow}, K])$

This protocol seems equivalent to the original one, but it is not so: here B can find out *already at step 1* if the first message does not comply with the definition, while in practice (and in the original definition) this can happen only after step 3. The proof of this is the fact that the modified protocol is *not flawed*: the above attack does not apply to it.

Concluding this section, checks are a simple yet crucial extension to the strand space models. The above protocol shows a situation where if checks are not used an attack might not be found.

It is worth mentioning that the protocol we reported here is not contrived, but it is actually similar to a subprotocol of the Zhou-Gollmann optimistic non-repudiation protocol [41] we have been working on for some time, where the same kind of check for testing the validity of M is carried out. In fact, the necessity of explicit checks became clear to us when we started using strand spaces for analyzing non-repudiation protocols. We don't report here the full [41] because non-repudiation protocols are outside the scope of this paper.

4 Monotonicity and Partial Runs

Let us fix some (not-too-formal) notation.

We say that a protocol has an (elementary) *authentication flaw* if there is a run in which one of the principals (e.g., the responder) at some points receives a message t while the other principal (e.g., the initiator) has not previously sent message t (this definition applies to the case of one session, for more sessions the extension is straightforward).

In addition, we say that a protocol presents a *secrecy flaw* if at some point a message that was to be kept secret can be known by the intruder. Both authentication and secrecy flaws can be checked with our tool; in particular, to check secrecy one can add the singleton strand $[-t]$ to the semibundle, where t is the message that should remain secret. If there is a run in which $-t$ is selected and "resolved", then the protocol has a flaw.

Thanks to its semantics, our system enjoys the following important monotonicity property.

Proposition 2 (Monotonicity). *Let S and S' be two semibundles such that $S \preceq S'$. Any authentication or secrecy flaw exhibited by S is exhibited by S' as well.*

This is of crucial practical importance. For instance, some vulnerabilities may only be found when considering partial runs. This is the case of the Woo and Lam mutual authentication protocol [40], which we introduce next. A possible attack of this protocol is described by Lowe [26]. The protocol aims at establishing a session key and provides mutual authentication between two agents A and B, with the help of a trusted server S.

Message 1. $A \rightarrow B : [A, N_A]$
Message 2. $B \rightarrow A : [B, N_B]$
Message 3. $A \rightarrow B : [A, B, N_A, N_B]^{\leftrightarrow}_{K_{AS}}$
Message 4. $B \rightarrow S : [A, B, N_A, N_B]^{\leftrightarrow}_{K_{AS}}, [A, B, N_A, N_B]^{\leftrightarrow}_{K_{BS}}$
Message 5. $S \rightarrow B : [B, N_A, N_B, K_{AB}]^{\leftrightarrow}_{K_{AS}}, [A, N_A, N_B, K_{AB}]^{\leftrightarrow}_{K_{BS}}$
Message 6. $B \rightarrow A : [B, N_A, N_B, K_{AB}]^{\leftrightarrow}_{K_{AS}}, [N_A, N_B]^{\leftrightarrow}_{K_{AB}}$
Message 7. $A \rightarrow B : [N_B]^{\leftrightarrow}_{K_{AB}}$

First A sends a nonce to B (message 1), and then B responds by returning another nonce (message 2). A then sends to B an encrypted message for the server (message 3). B forwards this message to the server, along with a similar message of his own (message 4).
When the server receives this message, he checks that the agent identities and nonces match. If so, the server selects a new key K_{AB} to be used in the subsequent session between A and B. Subsequently, the server creates two encrypted components, one for each of A and B, and returns both of them to B (message 5). B decrypts his component to obtain the key, and then forwards the other component to A, along with a component encrypted with K_{AB} (message 6). Finally, A decrypts his component to learn the key, and returns a message to B (message 7).

The Attack. The attack spans two sessions, α and β:

Message α.1. $I_A \rightarrow B : [A, B]$
Message α.2. $B \rightarrow I_A : [B, N_A]$
Message α.3. $I_A \rightarrow B :$ bit-string
Message α.4. $B \rightarrow I_S :$ bit-string, $[A, B, B, N_B]^{\leftrightarrow}_{K_{BS}}$
Message β.1. $I_A \rightarrow B : [A, N_B]$
Message β.2. $B \rightarrow I_A : [B, N'_B]$
Message β.3. $I_A \rightarrow B :$ bit-string$'$
Message β.4. $B \rightarrow I_S :$ bit-string$'$,$[A, B, N_B, N'_B]^{\leftrightarrow}_{K_{BS}}$
Message α.5. $I_S \rightarrow B :$ bit-string$''$,$[A, B, N_B, N'_B]^{\leftrightarrow}_{K_{BS}}$
Message α.6. $B \rightarrow I_A :$ bit-string$''$,$[B, N_B]^{\leftrightarrow}_{N'_B}$
Message α.7. $I_A \rightarrow B : [N_B]^{\leftrightarrow}_{N'_B}$

In message α.1, the intruder starts to impersonate A to attack B, *choosing B as the nonce.* B responds in message α.2 by returning a nonce N_B. Then, the intruder sends any bit-string of the appropriate length (message α.3). B creates an encrypted component of the appropriate form, and tries forwarding this to the server in message α.4, but the intruder intercepts this message. Then

the intruder starts a second run, β, again impersonating A, so as to use B as an oracle. He sends, in message $\beta.1$, the nonce N_B that B chose for run α. B returns a nonce (message $\beta.2$) and the intruder again responds with a bit-string (message $\beta.3$). When B tries to send message $\beta.4$ to the server, the intruder intercepts it again. This message is precisely of the form expected by B in message $\alpha.5$, hence the intruder forwards this message to B, and B accepts N'_B as a session key. Finally, B sends an appropriate message $\alpha.6$ and the intruder returns the component $[N_B]^{\leftrightarrow}_{N'_B}$ (message $\alpha.7$), which he can produce because he knows the "key" N'_B, thus completing the handshake.

Finding the attack. There are three protocol roles in the Woo and Lam protocol, A, B and the server S. The attack we are looking for spans two parallel sessions. In practice, we need a semibundle with at least two responder roles and no initiator roles, and it turns out that we don't have to specify any initiator nor server roles (adding one or more initiator or server roles to the semibundle would not prevent the system from finding the attack, however, it would slow it down; the same applies to the adding of more responder roles). The Prolog code is the following (we also specify the semibundle):

```
resp(A,B,N_A,N_N,K_AB,K_AS,K_BS,[
   recv([A,N_A]),
   send([B,N_B]),
   recv([A,[B,[N_A,N_B]]]+K_AS),
   send([([A,[B,[N_A,N_B]]]+K_AS),
         ([A,[B,[N_A,N_B]]]+K_BS)
        ]),
   recv([([B,[N_A,[N_B,K_AB]]]+K_AS),
         ([A,[N_A,[N_B,K_AB]]]+K_BS)
        ]),
   send([([B,[N_A,[N_B,K_AB]]]+K_AS),
         ([N_A,N_B]+K_AB)
        ]),
   recv(N_B+K_AB)
   ]).
semibundle([Sb1,Sb2]) :-
   resp(a,b,N_A1,n_B1,K_AB1,K_AS,k_BS,Sb1),
   resp(a,b,N_A2,n_B2,K_AB2,K_AS,k_BS,Sb2).
```

By running this code, we obtained Lowe's attack:

```
trace:
[responder(a, b, b, n_B1, n_B2, _1154, k_BS), recv([a,b])]
[responder(a, b, b, n_B1, n_B2, _1154, k_BS), send([b,n_B1])]
[responder(a, b, b, n_B1, n_B2, _1154, k_BS), recv([a, [b,
[b, n_B1]]] + _1154)]
[responder(a, b, b, n_B1, n_B2, _1154, k_BS), send([[a, [b,
[b, n_B1]]] + _1154, [a, [b, [b, n_B1]]] + k_BS])]
[responder(a, b, n_B1, n_B2, _1318, _1154, k_BS), recv([a,n_B1])]
[responder(a, b, n_B1, n_B2, _1318, _1154, k_BS), send([b,n_B2])]
```

```
[responder(a, b, n_B1, n_B2, _1318, _1154, k_BS), recv([a,
[b, [n_B1, n_B2]]] + _1154)]
[responder(a, b, n_B1, n_B2, _1318, _1154, k_BS), send([[a,
[b, [n_B1, n_B2]]] + _1154, [a, [b, [n_B1, n_B2]]] + k_BS])]
[responder(a, b, b, n_B1, n_B2, _1154, k_BS), recv([[b, [b,
[n_B1, n_B2]]] + _1154, [a, [b, [n_B1, n_B2]]] + k_BS])]
[responder(a, b, b, n_B1, n_B2, _1154, k_BS), send([[b, [b,
[n_B1, n_B2]]] + _1154, [b, n_B1] + n_B2])]
[responder(a, b, b, n_B1, n_B2, _1154, k_BS), recv(n_B1+n_B2)]

reached state:
[[responder(a, b, b, n_B1, n_B2, K_AS, k_BS), []],
[responder(a, b, n_B1, n_B2, K_AB2, K_AS, k_BS), [recv([[b,
[n_B1, [n_B2, K_AB2]]] + K_AS, [a, [n_B1, [n_B2, K_AB2]]] +
k_BS]), send([[b, [n_B1, [n_B2, K_AB2]]] + K_AS, [n_B1,
n_B2] + K_AB2]), recv(n_B2 + K_AB2)]]]
```

Here, it is worth noticing that the system chooses the term `recv([a, [b, [b, n_B1]]] + _1154)` as a possible bit-string of proper length (introduced by Lowe). Here, `_1154` is a variable, representing an infinite number of messages, all possible bit-strings. Also, notice that this is a partial run: in fact, the second strand is not completed. Inspecting the output, we can see that the system terminates right after message $\beta.4$.

The Original System. To emphasize the practical importance of Proposition 2 we now show what needs to be done to detect the above attack in the original system MS. In that context, the only way of finding the flaw is by employing *exactly* two responder strands, in which the second strand contains only and exactly the first four events:

```
responder(A,B,N_A,N_B,K_AB,K_AS,K_BS,[
   recv([A,N_A]),
   send([B,N_B]),
   recv([A,[B,[N_A,N_B]]]+K_AS),
   send([([A,[B,[N_A,N_B]]]+K_AS),
         ([A,[B,[N_A,N_B]]]+K_BS)
        ]),
   recv([([B,[N_A,[N_B,K_AB]]]+K_AS),
         ([A,[N_A,[N_B,K_AB]]]+K_BS)
        ]),
   send([([B,[N_A,[N_B,K_AB]]]+K_AS),
         ([N_A,N_B]+K_AB)
        ]),
   recv(N_B+K_AB)
 ]).

responder_incomplete(A,B,N_A,N_B,K_AB,K_AS,K_BS,[
   recv([A,N_A]),
   send([B,N_B]),
```

```
   recv([A,[B,[N_A,N_B]]]+K_AS),
   send([([A,[B,[N_A,N_B]]]+K_AS),
         ([A,[B,[N_A,N_B]]]+K_BS)])
 ]).
```

This is clearly *ad-hoc*.

5 Benchmarks

Comparison with MS. We use the Bilateral Key Exchange with Public Key protocol (BKEPK) (described in the Clark and Jacob library [16]), and checked secrecy of nonces. The protocol proceeds as follows:

Message 1. $B \to A : B, [N_B, B]^{\to}_{pk(A)}$
Message 2. $A \to B : [h(N_B), N_A, A, K_{AB}]^{\to}_{pk(B)}$
Message 3. $B \to A : [h(N_A)]^{\leftrightarrow}_{K_{ab}}$

Our test-bed was a Pentium IV 1.9GHz running Linux 2.4.7-10 and ECL^iPS^e 5.2. The comparison is shown in the following table. To make a fair comparison and avoid listing each partial run as a possible solution, we forced the system to find a secrecy flaw, which, since we are dealing with a non-flawed protocol, causes all possible partial runs to be explored.

# strands	description	Our system (time (s))	MS (time (s))
6	2A2B2T	0.38	3.20
7	3A2B2T	2.61	42.52
8	3A2B3T	2.84	324.22
7	3A3B1T	236.23	991.20
8	3A3B2T	277.23	6865.70
9	3A3B3T	316.20	57635.48

The description field gives an idea of the semibundle tested, e.g. 2A2B2T means that we are including 2 A roles, 2 B roles and checking secrecy of two nonces (one for each session).

Operationally speaking, the main difference between our system and MS lies in the fact that we check the constraints as soon as possible while producing the run. This is in line with our earlier paper [17] (although the system presented here has better performance). In MS first all (complete) runs are generated and then it is checked which runs yield a solvable constraint: in our system we anticipate failure, therefore reducing the search space. A more detailed comparison of the efficiency of both systems is future work.

Experiments with protocols from Clark and Jacob library. [16] We ran our system succesfully on several flawed protocols from the Clark and Jacob library [16]. The following table lists the performance and the attacks found.

Protocol	Attack	time (s)
Encrypted Key Exchange	Parallel Session	0.07
ISO symmetric key 1-pass unilateral auth	Replay	0.04
ISO public key 2-pass mutual authentication	Replay	0.11
Needham-Schroeder Symmetric Key	Man-in-the-middle	0.09
Needham-Schroeder Public Key	Man-in-the-middle	0.06
Needham-Schroeder-Lowe Public Key	Type Flaw	0.07
Neuman Stubblebine	Type Flaw	0.27
Otway-Rees	Type Flaw	0.22
SPLICE	Replay	0.21
Woo-Lam Mutual Authentication	Parallel Session	30.96
Yahalom with Lowe's alteration	Type Flaw	0.08

6 Conclusions

We have developed a constraint-based system for protocol verification based on the work of Millen and Shmatikov (MS) [30]. Our contribution is threefold:

- We have defined a new semantics which is more intuitive and yields to a considerably more efficient implementation, which can make the difference between days and minutes. This means that our system allows in practice the verification of more complex protocols and sessions than MS.
- We have a *monotonic system*: any (authentication or secrecy) attack that can be found using a semibundle S can be found also when using a semibundle S', provided that $S \preceq S'$.

  This has direct impact on the practicality of the system. For instance, our system allows for the automatic detection of attacks that rely on partial strands. While there are many such attacks (all attacks presented in this paper, for instance, rely on partial runs), the only way to detect them with MS is by providing the system with an ad-hoc semibundle that precisely reflects the partial run that exhibits the flaw.

  Suppose one wants to check seriously a protocol involving agents A and B for secrecy flaws, in presence of – say – maximal two parallel sessions. While with our system, thanks to Proposition 2, it is sufficient to run the tool on a semibundle S containing four honest strands (plus the secrecy strands), with MS one would have to try all semibundles $S' \preceq S$, which is infeasible.
- We have extended the system with the possibility of applying explicit checks. As shown in Section 3 this allows to model more complex protocols. In particular, we needed this feature when we used the system to analyze some non-repudiation protocols, among which Zhou-Gollmann's [41].

The system we presented here seems to be very fast. It is considerably faster than MS, which is well-known for its speed. In the near future, we intend to test its performances against other "fast" systems such as that of Chevalier and Vigneron [13] and of Basin [5].

7 Related Work

In previous work [17] we have defined a tool for protocol verification based on a *proof system*; the system presented here is radically different in syntax, semantics and implementation, and in particular it is much faster.

Several *logic*-based approaches have been applied to verification problems of security protocols. Among them, trace semantics is at the basis of the verification methods proposed, e.g., in [33,34,6,5]. In [34], Paulson models a protocol in presence of an intruder as an inductively defined set of traces, and uses Isabelle and HOL to prove *interactively* the absence of attacks. Paulson's approach works on an infinite-state search space. Our approach is actually closer to Basin's method [5], where *lazy data structures* are used to generate the infinite-search tree representing protocol traces in a *demand-driven* manner. To limit state explosion, however, Basin applies heuristics that prune the generated tree. Here – on the contrary – no heuristics is needed. Trace semantics were also used for model-checking based analysis. In this context, we find the work of Lowe [25,27], who first identified and fixed the flaws in the Needham-Schroeder protocol, and the work of Mitchell et al. [32]. As pointed out e.g., by Jacquemard, Rusinowitch and Vigneron [23], these approaches require ad-hoc solutions for limiting the search space. In contrast, our approach terminates for all protocols, provided that we limit the number of parallel sessions. Fiore and Abadi [20] and Boreale [8] also present algorithms for computing symbolic traces of infinite-state protocols. Automatic methods for rapid prototyping and analysis of protocols have also been presented. We mention Denker, Meseguer and Talcott [18] using Maude, and Mitchell, Mitchell and Stern [32] using Murϕ.

In a different approach, Cervesato et al. [11] use a formalism based on multiset-rewriting (linear logic) to specify protocol rules and actions of intruders. A multiset rewriting formalism has also been used by Rusinowitch et al. by implementing it in OCAML in the CASRUL system [13], and, in the later [23] by processing the rewrite rules in the theorem-prover daTac. Conceptually, our system shares some features with that developed (independently) by Chevalier and Vigneron [15,14]. As in Chevalier and Vigneron [15,14], we deal with communication in a constraint-based way, in which the intruder only checks if he is able to generate a certain message. On the other hand, Chevalier and Vigneron [15,14] rely on a multiset rewriting formalism.

In addition to MS a few works have appeared that employ constraint-solving techniques. Typically, the task of the intruder is brought down to that of checking whether certain messages are *contrivable*. Among these works we find Rusinowitch and Turuani [37] and Huima's [22]. In the former the authors also demonstrate that the problem of finding an attack *in the case of bounded parallel sessions* is NP-complete. In our work, we require the intruder only to *check* whether he is able to generate a certain message, while these approaches do this in a more radical way: the checking phase is postponed *after* that the whole trace has been completed (we check it after the next protocol step has been carried out).

Logic programming languages have been applied to specify security protocols in several different ways. Meadow's NRL Protocol Analyzer [29] performs

a reachability analysis using state-enumeration enriched by lemmata proved by induction. This way, NRL can cope with a potential infinite search space. Our approach differs from this previous work in the first place in that NRL explores the search space in a backward fashion, i.e., starting from the unwanted situation and working backwards to find possible traces leading to it.

A logic programming language is used by Aiello and Massacci [3] but using a different perspective, i.e., with *stable semantics*, to specify and debug protocols. In this setting knowledge, protocol rules, intruder capabilities and objectives are specified in a declarative way. Finally, Blanchet [7] uses Prolog to specify *conservative abstractions* that can be used to prove security protocols free from attacks. Intuitively, Horn clauses are used here as *constructors* and *destructors* for the messages exchanged on the net and intercepted by the intruder. Abadi and Blanchet [2] relate this approach to that of using *types* for guaranteeing the secrecy of communication [1], an approach substantially different from ours. The verification procedure combines aspects of *unfolding* and *bottom-up* evaluation (following, however, a depth-first strategy). A way of better understanding the relationships with our approach would be to extend the bottom-up evaluation strategy to the multi-conclusion setting as suggested by Bozzano [9].

Most of the above work can analyze a finite number of protocol sessions. Noteworthy exception to this rule are: firstly, the approaches based on Theorem Proving (Paulson [33,34]), together with the NRL analyzer, that require human intervention to limit the search space. Secondly, the works based on some form of abstraction, which does not guarantee completeness, like Abadi and Blanchet [2] (which is based on types) and Schneider's work on *rank functions* [38]. Blanchet also developed a protocol verifier [7], which guarantees termination but has the disadvantage that false attacks may be reported. Finally, there is work in which the analysis might not terminate, like Basin [5] (however the heuristics they apply are successful in most cases).

Acknowledgments. We want to thank Pieter Hartel, Jonathan Millen, Vitaly Shmatikov, and the anonymous referees for their useful comments.

References

1. M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In F. Honsel and M. Miculan, editors, *Proc. Foundation of Software Science and Computation Structures (FoSSaCS 2001)*, volume 2030 of *LNCS*, pages 25–41. Springer-Verlag, 2001.
2. M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. In *29th ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 33–44, Portland, Oregon, 2002. ACM Press.
3. L. C. Aiello and F. Massacci. Verifying security protocols as planning in logic programming. *Transactions on Computational Logic*, 2(4):542–580, 2001.
4. K. R. Apt. *From Logic Programming to Prolog.* International Series in Computer Science. Prentice Hall, 1997.

5. D. Basin. Lazy infinite-state analysis of security protocols. In R. Baumgart, editor, *Secure Networking - CQRE (Secure) '99, International Exhibition and Congress*, volume 1740 of *LNCS*, pages 30–42. Springer-Verlag, 1999.
6. G. Bella and L. C. Paulson. Kerberos version IV: Inductive analysis of the secrecy goals. In J.-J. Quisquater, editor, *Proc. 5th European Symposium on Research in Computer Security*, volume 1485 of *LNCS*, pages 361–375, Louvain-la-Neuve, Belgium, 1998. Springer-Verlag.
7. B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In S. Schneider, editor, *Proc. 14th IEEE Computer Security Foundations Workshop*, 2001.
8. M. Boreale. Symbolic trace analysis of cryptographic protocols. In *28th Colloquium on Automata, Languages and Programming (ICALP)*, LNCS, pages 667–681. Springer-Verlag, 2001.
9. M. Bozzano. Ensuring security through model checking in a logical environment (preliminary results). In G. Delzanno, S. Etalle, and M. Gabbrielli, editors, *Proc. Workshop on Specification, Analysis and Validation for Emerging Technologies (SAVE01)*, 2001.
10. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.
11. I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *PCSFW: Proc. 12th Computer Security Foundations Workshop*, pages 55–69. IEEE Computer Society Press, 1999.
12. I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *PCSFW: Proc. 13th Computer Security Foundations Workshop*, pages 35–51. IEEE Computer Society Press, 2000.
13. Y. Chevalier, F. Jacquemard, M. Rusinowitch, M. Turuani, and L. Vigneron. CASRUL web site. `http://www.loria.fr/equipes/protheo/SOFTWARES/CASRUL/`.
14. Y. Chevalier and L. Vigneron. A tool for lazy verification of security protocols. In *Proc. 16th IEEE International Conference Automated Software Engineering*, 2001.
15. Y. Chevalier and L. Vigneron. Towards efficient automated verification of security protocols. In *Proc. VERIF01, Verification Workshop in conjunction with IJCAR*, pages 19–33, 2001.
16. J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. `http://www.cs.york.ac.uk/ jac/papers/drareview.ps.gz`, 1997.
17. G. Delzanno and S. Etalle. Proof theory, transformations, and logic programming for debugging security protocols. In A. Pettorossi, editor, *Proc. Eleventh International Workshop on Logic Program Synthesis and Transformation – LOPSTR 2001*, LNCS, pages 76–91. Springer-Verlag, 2002.
18. G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In N. Heintze and J. Wing, editors, *In Proc. of Workshop on Formal Methods and Security Protocols*, 1998.
19. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
20. M. Fiore and M. Abadi. Computing symbolic models for verifying cryptographic protocols, 2001.
21. J.C. Herzog F.T. Fabrega and J.D. Guttman. Strand spaces: Why is a security protocol correct? In *Proceedings of The 1998 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1998.
22. A. Huima. Efficient infinite-state analysis of security protocols. In *Proc. Workshop Formal Methods and Security Protocols (FLOC 1999)*, 1999.

23. F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and verifying security protocols. In M. Parigot and A. Vonkorov, editors, *Proc. LPAR: International Conference on Logic for Programming and Automated Reasoning*, number 1995 in LNCS, pages 131–160. Springer-Verlag, 2000.
24. J. W. Lloyd. *Foundations of Logic Programming.* Symbolic Computation – Artificial Intelligence. Springer-Verlag, Berlin, 1987. Second edition.
25. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055, pages 147–166. Springer-Verlag, Berlin Germany, 1996.
26. G. Lowe. Some new attacks upon security protocols. In *PCSFW: Proceedings of The 9th Computer Security Foundations Workshop.* IEEE Computer Society Press, 1996.
27. G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW '97)*, pages 18–30. IEEE, 1997.
28. C. Meadows. Formal verification of cryptographic protocols: A survey. In J. Pieprzyk and R. Safavi-Naini, editors, *Advances in Cryptology – ASIACRYPT ' 94*, LNCS, pages 133–150. Springer-Verlag, 1995.
29. C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
30. J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proc. 2001 ACM Conference on Computer and Communication Security*, pages 166 – 175. ACM press, 2001.
31. J. K. Millen, S. C. Clark, and S. B. Freedman. The Interrogator: Protocol security analysis. *IEEE Transactions on Software Engineering*, 13(2):274–288, February 1987. Special Issue on Computer Security and Privacy.
32. J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murϕ. In *Proceedings of the 1997 Conference on Security and Privacy*, pages 141–153. IEEE Press, 1997.
33. L. C. Paulson. Mechanized proofs of security protocols: Needham-Schroeder with public keys. Technical Report 413, University of Cambridge, Computer Laboratory, January 1997.
34. L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
35. A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *IEEE Symposium on Foundations of Secure Systems*, 1995.
36. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1999.
37. M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is np-complete. In S. Schneider, editor, *Proc. 14th IEEE Computer Security Foundations Workshop*, 2001.
38. P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. Modelling and analysis of security protocols, 2001.
39. D. X. Song, S. Berezin, and A. Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1/2):47–74, 2001.
40. T.Y.C. Woo and S. S. Lam. A lesson on authenticated protocol design. *Operating Systems Review*, 28(3):24–37, 1994.
41. J. Zhou and D. Gollmann. An efficient non-repudiation protocol. In *Proceedings of the 10th Computer Security Foundations Workshop (CSFW '97)*, pages 126–132, 1997.

From Secrecy to Authenticity in Security Protocols*

Bruno Blanchet[1,2]

[1] Département d'Informatique
École Normale Supérieure, Paris
[2] Max-Planck-Institut für Informatik
Bruno.Blanchet@ens.fr

Abstract. We present a new technique for verifying authenticity in cryptographic protocols. This technique is fully automatic, it can handle an unbounded number of sessions of the protocol, and it is efficient in practice. It significantly extends a previous technique for the verification of secrecy. The protocol is represented in an extension of the pi calculus with fairly arbitrary cryptographic primitives. This protocol representation includes the authentication specification to be verified, but no other annotation. Our technique has been proved correct, implemented, and tested on various protocols from the literature. The experimental results show that we can verify these protocols in less than 1 s.

1 Introduction

The verification of cryptographic protocols has already been the subject of numerous research works. It is indeed particularly important since flaws have been found in many published protocols. A recent trend in this area aims to verify protocols with an unbounded number of sessions, while relying as little as possible on human intervention. This goal is achieved using language-based techniques such as typing or abstract interpretation, that can handle infinite-state systems thanks to safe approximations. These techniques are not complete (a correct protocol can fail to typecheck, or false attacks can be found by abstract interpretation tools), but they are sound (when they do not find attacks, the protocol is guaranteed to satisfy the considered property). This is important for the certification of protocols.

Our goal in this paper is to extend previous work in this line of research by providing a fully automatic technique to verify authenticity in cryptographic protocols, without bounding the number of sessions of the protocol. Intuitively, a protocol authenticates A to B if, when B thinks he talks to A, then he actually talks to A. A simple and widely used definition of authenticity is the formalization by a correspondence property [27,19], according to the following scheme. When B thinks he has run the protocol with A, he emits a special event `end`.

* This work was supported in part by the RTD project IST-1999-20527 DAEDALUS of the european IST FP5 programme.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 342–359, 2002.
© Springer-Verlag Berlin Heidelberg 2002

When A thinks she runs the protocol with B, she emits another event `begin`. Authenticity is satisfied when B cannot emit his `end` event without A having emitted her `begin` event. There are several variations along this scheme in the literature, and, as we show in the following of the paper, our technique can handle most of them. Our technique is based on a substantial extension of our previous verification technique for secrecy [1,6]. (This technique for secrecy is also similar to that of [25].) We show that in some cases, a proof of secrecy for a modified protocol can directly give a proof of authenticity for the considered protocol. Then, we show how to extend our secrecy proof technique to handle the general case of authenticity.

More precisely, the protocol is represented in the process calculus introduced in [1], which is an extension of the pi calculus with fairly arbitrary cryptographic primitives. This process calculus is extended with instructions to emit `begin` and `end` events, to be able to represent the authenticity specification of the protocol. These events are the only required annotation of the protocol; no annotation is needed to help the tool proving authenticity.

The protocol is then automatically translated into a set of Horn clauses. This translation requires significant extensions with respect to the translation for secrecy given in [1], and can be seen as an implementation of a type system, as in [1]. Some of these extensions improve the precision of the analysis, in particular to avoid merging different names. Other extensions define the translation of events. Finally, this set of Horn clauses is passed to a solving algorithm, described in [6,25]. This solver does not always terminate, but we conjecture that it terminates for a large class of well-designed protocols (see Sect. 4.3). Our experiments also demonstrate that it terminates on many examples of protocols. A minor extension of this solver is required to prove authenticity.

The main advantages of our method could be summarized as follows. It is fully automatic; the user only has to code the protocol and the authentication specification. It puts no bounds on the number of sessions of the protocol, or the size of terms that the adversary can manipulate. It can handle fairly general cryptographic primitives, including shared-key encryption, public-key encryption, signatures, one-way hash functions, and Diffie-Hellman key agreements. It relies on a precise semantic foundation. One limitation of the technique is that, in rare cases, the solving algorithm does not terminate. The technique is also not complete: the translation into Horn clauses introduces approximations that we describe in Sect. 4.2. The tool provides sufficient conditions for proving authenticity, but can fail on correct protocols, even if that did not happen in our experiments.

Related work. We mainly focus on the few works that automatically verify authenticity in cryptographic protocols, without bounding the number of sessions.

Gordon and Jeffrey define a type system for verifying authenticity in cryptographic protocols, first for shared-key cryptography [14], then for public-key cryptography [15]. Our system allows more general cryptographic primitives (including hash functions and Diffie-Hellman key agreements), and more flexibility (for example, our system can verify the original Otway-Rees protocol [21]). Also, in our system, no annotation is needed, whereas in [14,15], explicit type casts

and checks have to be manually added. However, their system has the advantage that type checking always terminates, whereas in some rare cases, our analyzer does not.

Debbabi et al. [13] also verify authenticity thanks to a representation of protocols by inference rules, very similar to our Horn clauses. However, they verify a weaker notion of authenticity (corresponding to aliveness: if B terminates the protocol, then A must have been alive at some point before), and handle only shared-key cryptography.

A few other methods require little human effort, while supporting an unbounded number of runs: the verifier of [16], based on rank functions, can prove the correctness of or find attacks against protocols with atomic symmetric or asymmetric keys. Theorem proving [22] often requires manual intervention of the user. An exception to this is [12], but it deals only with secrecy. The theorem prover TAPS [11] often succeeds without or with little human intervention.

Model checking [18] in general implies a limit on the number of sessions of the protocol. This problem has been tackled by [7,23,8]. They recycle nonces, to use only a finite number of them in an infinite number of runs. The technique was first used for sequential runs, then generalized to parallel runs in [8], but with the additional restriction that the agents must be "factorisable". (Essentially, a single run of the agent has to be split into several runs such that each run contains only one fresh value.) Athena [24] uses strand spaces to reduce the state space, but still sometimes limits the number of sessions to guarantee termination.

Amadio and Prasad [4] already note that authenticity can be translated into secrecy, by using a judge process. The translation is limited in that only one message can be registered by the judge, so the verified authenticity property is not exactly the same as ours.

Outline. Section 2 introduces our process calculus. Section 3 defines secrecy, and the various notions of authenticity that we can verify. Section 4 explains and proves our verification technique for authenticity, and relates it to the verification of secrecy. Most proofs are sketched or omitted because of space constraints. Section 5 gives our experimental results on a selection of cryptographic protocols of the literature.

2 The Process Calculus

Figure 1 gives the syntax of terms (data) and processes (programs) of our calculus. The identifiers a, b, c, k, and similar ones range over names, and x, y, and z range over variables. The syntax also assumes a set of symbols for constructors and destructors; we often use f for a constructor and g for a destructor.

Constructors are used to build terms. Therefore, the terms are variables, names, and constructor applications of the form $f(M_1, \ldots, M_n)$. On the other hand, destructors do not appear in terms, but only manipulate terms in processes. They are partial functions on terms that processes can apply. The process *let* $x = g(M_1, \ldots, M_n)$ *in* P *else* Q tries to evaluate $g(M_1, \ldots, M_n)$; if this succeeds, then x is bound to the result and P is executed, else Q is executed. More

$M, N ::=$	terms
x, y, z	variable
a, b, c, k	name
$f(M_1, \ldots, M_n)$	constructor application
$P, Q ::=$	processes
$\overline{M}\langle N\rangle.P$	output
$M(x).P$	input
0	nil
$P \mid Q$	parallel composition
$!P$	replication
$(\nu a)P$	restriction
$\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$	destructor application
$\mathtt{begin}(M).P$	begin event
$\mathtt{end}(M).P$	end event
$\mathtt{begin\_ex}(M)$	executed begin event
$\mathtt{end\_ex}(M)$	executed end event

Fig. 1. Syntax of the process calculus

precisely, the semantics of a destructor g of arity n is given by a set $\mathrm{def}(g)$ of reduction rules of the form $g(M_1, \ldots, M_n) \rightarrow M$ where $M_1, \ldots, M_n, M$ are terms without free names. We extend these rules by $g(M'_1, \ldots, M'_n) \rightarrow M'$ if and only if there exists a substitution σ and a reduction rule $g(M_1, \ldots, M_n) \rightarrow M$ in $\mathrm{def}(g)$ such that $M'_i = \sigma M_i$ for all $i \in \{1, \ldots, n\}$, and $M' = \sigma M$. We assume that the set $\mathrm{def}(g)$ is finite (it usually contains one or two rules in examples). Note that destructors are defined by reduction rules instead of the equalities used in [1]. This allows destructors to yield several different results non-deterministically. This extension is used in our modeling of Diffie-Hellman agreements[1]. Using constructors and destructors, we can represent data structures and cryptographic operations as summarized in Fig. 2.

The process calculus provides instructions for emitting events, that will be used in authenticity specifications. The process $\mathtt{begin}(M).P$ emits the event $\mathtt{begin}(M)$, then executes P. The process $\mathtt{end}(M).P$ emits the event $\mathtt{end}(M)$, then executes P. The constructs $\mathtt{begin\_ex}(M)$ and $\mathtt{end\_ex}(M)$ are useful to remember that the events $\mathtt{begin}(M)$ and $\mathtt{end}(M)$ respectively have been executed. They do not appear in processes at the beginning, but are introduced by reductions. We can show that, in our semantics, the number of executions `begin` is equal to the number of occurrences of `begin_ex` in the reduced process, and similarly for `end`.

The other constructs in the syntax of Fig. 1 come from the pi calculus. The input process $M(x).P$ inputs a message on channel M, and executes P with x bound to the input message. The output process $\overline{M}\langle N\rangle.P$ outputs the message

[1] As explained in [2,6], the Diffie-Hellman key agreement can be modeled using the equation $f(x, g(y)) = f(y, g(x))$. This equation does not fit directly in the framework of constructors and destructors, but it can be simulated by using two constructors g and h and a destructor f defined by $f(x, y) \rightarrow h(x, y)$ and $f(x, g(y)) \rightarrow h(y, g(x))$.

Tuples:
Constructor: tuple $ntuple(M_1, \ldots, M_n)$
Destructors: projections $ith_n(ntuple(M_1, \ldots, M_n)) \rightarrow M_i$
Shared-key encryption:
Constructor: encryption of M under the key N, $sencrypt(M, N)$
Destructor: decryption $sdecrypt(sencrypt(M, N), N) \rightarrow M$
Public-key encryption:
Constructors: encryption of M under the public key N, $pencrypt(M, N)$
public key generation from a secret key N, $pk(N)$
Destructor: decryption $pdecrypt(pencrypt(M, pk(N)), N) \rightarrow M$
Signatures:
Constructor: signature of M with the secret key N, $sign(M, N)$
Destructors: signature verification $checksign(sign(M, N), pk(N)) \rightarrow M$
message without signature $getmessage(sign(M, N)) \rightarrow M$
One-way hash functions:
Constructor: hash function $H(M)$.

Fig. 2. Constructors and destructors

N on the channel M and then executes P. Here, we use an arbitrary term M to represent a channel, but the process blocks if M does not reduce to a name at runtime. The nil process 0 does nothing. The process $P \mid Q$ is the parallel composition of P and Q. The replication $!P$ represents an unbounded number of copies of P in parallel. The restriction $(\nu a)P$ creates a new name a, and then executes P. Moreover, we define *let* $x = M$ *in* P as a syntactic sugar for $P\{M/x\}$, and *if* $M = N$ *then* P *else* Q as a shorthand for *let* $x = equal(M, N)$ *in* P *else* Q, where the destructor *equal* is defined by $equal(M, M) \rightarrow M$. As usual, we may omit an *else* clause when it consists of 0.

The name a is bound in the process $(\nu a)P$. The variable x is bound in P in the processes $M(x).P$, *let* $x = g(M_1, \ldots, M_n)$ *in* P *else* Q. We write $fn(P)$ and $fv(P)$ for the sets of names and variables free in P, respectively. A process is closed if it has no free variables; it may have free names. We identify processes up to renaming of bound names and variables. We write $\{M_1/x_1, \ldots, M_n/x_n\}$ for the substitution that replaces $x_1, \ldots, x_n$ with $M_1, \ldots, M_n$, respectively. A starting process is a process that does not contain executed events.

We extend the standard semantics to handle authenticity (Fig. 3).

As a running example, we consider a simplified version of the Woo and Lam one-way public-key authentication protocol, version of [28], in which host names are replaced by public keys, which makes interaction with a server useless. (The version tested in the benchmarks is the full version.) The protocol is:

$$\begin{array}{l}\text{Message 1. } A \rightarrow B : pk_A \\ \text{Message 2. } B \rightarrow A : b \\ \text{Message 3. } A \rightarrow B : \{pk_A, pk_B, b\}_{sk_A}\end{array}$$

A first sends to B its public key, B replies with a nonce (fresh name), and A sends its public key, the public key of B, and the nonce all signed with its private key sk_A. This protocol can be represented in our calculus by the process P:

$$P \mid 0 \equiv P$$
$$P \mid Q \equiv Q \mid P$$
$$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$$
$$(\nu a_1)(\nu a_2)P \equiv (\nu a_2)(\nu a_1)P$$
$$(\nu a)(P \mid Q) \equiv P \mid (\nu a)Q \quad \text{if } a \notin fn(P)$$

$$P \equiv Q \;\Rightarrow\; P \mid R \equiv Q \mid R$$
$$P \equiv Q \;\Rightarrow\; !P \equiv !Q$$
$$P \equiv Q \;\Rightarrow\; (\nu a)P \equiv (\nu a)Q$$
$$P \equiv P$$
$$Q \equiv P \;\Rightarrow\; P \equiv Q$$
$$P \equiv Q,\, Q \equiv R \;\Rightarrow\; P \equiv R$$

$$\overline{a}\langle M\rangle.Q \mid a(x).P \;\rightarrow\; Q \mid P\{M/x\} \qquad \text{(Red I/O)}$$
$$\textit{let } x = g(M_1, \ldots, M_n) \textit{ in } P \textit{ else } Q \rightarrow P\{M'/x\}$$
$$\text{if } g(M_1, \ldots, M_n) \rightarrow M' \qquad \text{(Red Destr 1)}$$
$$\textit{let } x = g(M_1, \ldots, M_n) \textit{ in } P \textit{ else } Q \rightarrow Q$$
$$\text{if there exists no } M' \text{ such that } g(M_1, \ldots, M_n) \rightarrow M' \qquad \text{(Red Destr 2)}$$
$$!P \;\rightarrow\; P \mid !P \qquad \text{(Red Repl)}$$
$$\texttt{begin}(M).P \;\rightarrow\; \texttt{begin\_ex}(M) \mid P \qquad \text{(Red Begin)}$$
$$\texttt{end}(M).P \;\rightarrow\; \texttt{end\_ex}(M) \mid P \qquad \text{(Red End)}$$
$$P \;\rightarrow\; Q \;\Rightarrow\; P \mid R \;\rightarrow\; Q \mid R \qquad \text{(Red Par)}$$
$$P \;\rightarrow\; Q \;\Rightarrow\; (\nu a)P \;\rightarrow\; (\nu a)Q \qquad \text{(Red Res)}$$
$$P' \equiv P,\, P \;\rightarrow\; Q,\, Q \equiv Q' \;\Rightarrow\; P' \;\rightarrow\; Q' \qquad \text{(Red} \equiv\text{)}$$

Fig. 3. Structural congruence and reduction

$$P_A(sk_A, pk_A) = c(x\_pk_B).\texttt{begin}(x\_pk_B).\overline{c}\langle pk_A\rangle.c(x\_b).$$
$$\quad \overline{c}\langle sign((pk_A, x\_pk_B, x\_b), sk_A)\rangle$$
$$P_B(sk_B, pk_B, pk_A) = c(x\_pk_A).(\nu b)\overline{c}\langle b\rangle.c(m).$$
$$\quad \textit{let } (y\_pk_A, y\_pk_B, y\_b) = \textit{checksign}(m, x\_pk_A) \textit{ in}$$
$$\quad \textit{if } y\_pk_A = x\_pk_A \textit{ then if } y\_pk_B = pk_B \textit{ then if } y\_b = b \textit{ then}$$
$$\quad \textit{if } x\_pk_A = pk_A \textit{ then } \texttt{end}(pk_B)$$
$$P = (\nu sk_A)(\nu sk_B)\textit{let } pk_A = pk(sk_A) \textit{ in let } pk_B = pk(sk_B) \textit{ in}$$
$$\quad \overline{c}\langle pk_A\rangle.\overline{c}\langle pk_B\rangle.((!P_A(sk_A, pk_A)) \mid (!P_B(sk_B, pk_B, pk_A)))$$

The channel c is public: The adversary can send and listen on it. We use a single public channel and not two or more channels because the adversary could take a message from one channel and relay it on another channel, thus removing any difference between the channels. The process P begins with the creation of the secret and public keys of A and B. The public keys are output on channel c to model that the adversary has them in its initial knowledge. Then the protocol itself starts: P_A represents A, P_B represents B. Both principals can run an unbounded number of sessions, hence the replications. We consider that A and B are both willing to talk to any principal. So, to determine to whom A will talk, we consider that A first inputs a message containing the public key of its interlocutor (this interlocutor is therefore chosen by the adversary). Then A starts the protocol by emitting a begin event $\texttt{begin}(x\_pk_B)$, whose intuitive meaning is "A has started a session with the host of public key x_pk_B". At the

end of P_B, when B thinks he talks with A (that is, when $x\_pk_A = pk_A$), he emits an end event $\texttt{end}(pk_B)$, whose meaning is "B thinks he has completed a session with A".[2] If the protocol is correct, B should only emit $\texttt{end}(pk_B)$ when A has first emitted $\texttt{begin}(pk_B)$. This is what we are going to formalize and check in the following.

3 Definitions of Secrecy and Authenticity

We assume that the protocol is executed in the presence of an adversary that can listen to all messages, compute, and send all messages it has, following Dolev and Yao. Thus, an adversary is any process that has a set of public names S in its initial knowledge.

Definition 1. *Let S be a finite set of names. The closed process Q is an S-*adversary *if and only if $fn(Q) \subseteq S$ and Q does not contain* `begin`*,* `end`*,* `begin_ex`*, or* `end_ex` *events.*

Secrecy. We adopt the definition of secrecy of [1].

Definition 2. *We say that the closed process P* outputs *M on c if and only if $P\ (\rightarrow \cup \equiv)^*\ \overline{c}\langle M\rangle.R \mid R'$ for some processes R and R'. The closed process P* preserves the secrecy *of M from S if and only if $P \mid Q$ does not output M on c for any S-adversary Q and any $c \in S$.*

Non-injective agreement. Non-injective agreement means that if an event $\texttt{end}(M)$ is executed, then $\texttt{begin}(M)$ has also been executed.

Definition 3. *The closed starting process P satisfies* non-injective agreement *with respect to S-adversaries if and only if for any S-adversary Q, for any P' such that $P \mid Q\ (\rightarrow \cup \equiv)^*\ P'$, for any M, if $\texttt{end\_ex}(M)$ occurs in P', then $\texttt{begin\_ex}(M)$ also occurs in P'. (In P', the restrictions are renamed such that they bind pairwise different names, and names different from free names.)*

Intuitively, in the example of Sect. 2, when non-injective agreement is satisfied, B cannot emit an `end` event without A having executed a `begin` event, so B cannot terminate the protocol thinking he talks to A, without A being actually involved in the protocol. Moreover, the agreement on the parameter of the events, $x\_pk_B = pk_B$, implies that A actually thinks she talks to B.

By choosing different values for the parameter M, the definition can encode some of Lowe's authentication properties [19] (non-injective agreement, aliveness, and a slightly stronger version of weak agreement).

[2] Note that the end event must not be emitted when B thinks he talks to the adversary. Indeed, in this case, it is correct that no begin event has been emitted by the interlocutor of B, since the adversary never emits events.

Injective agreement. Injective agreement furthermore requires that the number of executions of $\texttt{end}(M)$ is smaller than the number of executions of $\texttt{begin}(M)$.

Definition 4. *The closed starting process P satisfies* injective agreement *with respect to S-adversaries if and only if for any S-adversary Q, for any P' such that $P \mid Q\ (\rightarrow \cup \equiv)^*\ P'$, for any M, the number of occurrences of* $\texttt{end\_ex}(M)$ *in P' is at most the number of occurrences of* $\texttt{begin\_ex}(M)$ *in P'. (In P', the restrictions are renamed such that they bind pairwise different names, and names different from free names.)*

This definition corresponds to Lowe's agreement specification [19], and to Woo and Lam's correspondence assertions [27].

4 Verification

4.1 A First Idea

With the additional assumption that, at runtime, *when the events* $\texttt{begin}(M)$ *or* $\texttt{end}(M)$ *are executed, M does not contain bound names*, it is easy to prove authenticity by using secrecy, as shown by the following theorem:

Theorem 1. *Let $P = C[\texttt{begin}(M).P_1, \texttt{end}(M').P_2]$ be a closed starting process (where $C[]$ is any context), and assume that these are the only* $\texttt{begin}$ *and* $\texttt{end}$ *events in P. Let $P'(N) = C[\textit{if } M = N \textit{ then } 0 \textit{ else } P_1, \overline{c}\langle \mathrm{auth}(M')\rangle.P_2]$, where* auth *is a new function symbol. Assume that for all closed terms N, $P'(N)$ preserves the secrecy of* $\mathrm{auth}(N)$ *from S (with $c \in S$). Then P satisfies non-injective agreement with respect to S-adversaries.*

Intuitively, we use the following proof strategy : we fix a set $E_b = \{\texttt{begin}(M) \mid M \not\models N\}$ (for any N) of allowed $\texttt{begin}$ events, and we modify the process P such that, if it tries to execute a $\texttt{begin}$ event not in E_b, it blocks. We show that the modified process cannot execute $\texttt{end}(N)$. Indeed, when P tries to execute $\texttt{begin}(N)$, P' executes 0; and since P' preserves the secrecy of $\mathrm{auth}(N)$, it cannot execute $\overline{c}\langle \mathrm{auth}(N)\rangle$, so P cannot execute $\texttt{end}(N)$. This implies non-injective agreement: if $\texttt{begin}(N)$ cannot be executed, then $\texttt{end}(N)$ cannot be executed.

Then, we could apply standard techniques for checking secrecy, for example that of [25,6,1], with only very minor extensions. However, the condition that bound names do not appear in events introduces important limitations: for example, we cannot check the agreement on a session key, nor injective agreement.

Our work in the following of this paper will explain how we can overcome these limitations. First, Theorem 1 has to be modified: in a trace that executes $\texttt{end}(N)$, N may contain bound names whose scope does not include $\texttt{begin}(M)$ in the initial process, so the term N cannot be written in the test *if* $M = N$ *then* 0 *else* P_1 in place of the $\texttt{begin}(M)$ event. Fortunately, in the checker of [25,6], bound names are represented as functions of the inputs that precede their creation, and these functions have no scope limit. So we are going to use this representation of bound names in our translation into Horn clauses.

Second, when translating the process into Horn clauses, we have to make sure that different terms are represented by different values in the checker, so that the else branch of the test $M = N$ can be precisely analyzed. (If different terms are merged in the checker, the checker can never be sure that M is equal to N, and so it must consider that the else branch may always be taken. This leads to an unacceptable loss of precision for the analysis of authenticity.) This is not true in the presence of bound names in the checkers of [25,6], since names created by the same restriction after receiving the same inputs are merged. This merging can be avoided by adding a parameter to names, that we call a *session identifier*. Such a session identifier indicates which copy of the restriction created the name, and different copies will have different session identifiers.

Third, to prove injective agreement, we have to relate corresponding sessions of the two participants of the protocol. This is also done using session identifiers.

4.2 Translation into Horn Clauses

We consider a closed starting process P_0 representing the protocol to check. We assume that different variables and names have different identifiers in P_0. (This can be achieved by renaming.) We also assume that P_0 contains a single occurrence of `begin` and a single occurrence of `end`. In most cases, when an authenticity specification contains several `begin` and `end` events, each occurrence of `end` should correspond to a fixed occurrence of `begin`. A constant element in the parameters of `begin` and `end` can enforce this property: $\texttt{end}(c_1, \ldots)$ corresponds to $\texttt{begin}(c_1, \ldots)$, $\texttt{end}(c_2, \ldots)$ to $\texttt{begin}(c_2, \ldots)$, and so on. The implementation supports this case, by verifying each correspondence independently: it first checks the correspondence between $\texttt{begin}(c_1, \ldots)$ and $\texttt{end}(c_1, \ldots)$, then for c_2, etc.

Given the closed starting process P_0 and a set of names S, the protocol checker builds a set of Horn clauses, representing the protocol in parallel with any S-adversary. These clauses contain "patterns" ranged over by p, p' and generated by the grammar:

$p ::=$	patterns
x, y, z, i	variable
$a[p_1, \ldots, p_n, i_1, \ldots, i_{n'}]$	name
$f(p_1, \ldots, p_n)$	constructor application

For each name a in P_0 we have a corresponding pattern construct $a[p_1, \ldots, p_n, i_1, \ldots, i_{n'}]$. We treat a as a function symbol, and write $a[p_1, \ldots, p_n, i_1, \ldots, i_{n'}]$ rather than $a(p_1, \ldots, p_n, i_1, \ldots, i_{n'})$ only to distinguish names from constructors. If a is a free name, then the arity of this function is 0. If a is bound by a restriction $(\nu a)P$ in P_0, then this arity is the total number of input statements, destructor applications, and replications above the restriction $(\nu a)P$ in the abstract syntax tree of P_0. The first parameters of $a[\ldots]$ are patterns $p_1, \ldots, p_n$ corresponding to input statements and destructor applications. The last parameters of $a[\ldots]$ are session identifiers $i_1, \ldots, i_{n'}$ that are integers or variables (taken in a set V_s disjoint from the set V_o of ordinary variables). There is one session identifier for each replication above the restriction (νa). Closed patterns are named types, and denoted by T.

The clauses use 4 predicates: attacker, message, begin, and end. The fact attacker(p) means that the attacker may have p, message(p, p') that the message p' may appear on channel p, begin(p, ρ) that the event `begin` has been executed with a parameter corresponding to p and an environment ρ, and end(p, s) that `end` has been executed in session s with a parameter corresponding to p.

$F ::=$	facts
attacker(p)	attacker knowledge
message(p, p')	channel messages
begin(p, ρ)	begin event
end(p, s)	end event

The rules comprise rules for the attacker and rules for the protocol, defined below. These rules form the set $B_{P_0,S}$. The predicate begin(p, ρ) is defined by a set of closed facts B_b, that do not belong to $B_{P_0,S}$. The set B_b corresponds to the translation into Horn clauses of the set E_b used in the proof of Theorem 1.

Rules for the attacker. The rules describing the attacker are essentially the same as for the verification of secrecy in [1]. The only difference is that, here, the attacker is given an infinite set of fresh names $b[i]$, instead of only one fresh name $b[]$. Indeed, we cannot merge all fresh names created by the attacker, since we have to make sure that different terms are represented by different patterns for the authenticity test to be correctly implemented, as seen in Sect. 4.1. The abilities of the attacker are then represented by the following rules:

$$\text{For each } a \in S, \text{ attacker}(a[]) \tag{Init}$$

$$\text{attacker}(b[i]) \text{ where } b \notin fn(P_0) \text{ and } (\nu b) \text{ does not occur in } P_0 \tag{Rn}$$

$$\begin{array}{l}\text{For each constructor } f \text{ of arity } n,\\ \quad \text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n))\end{array} \tag{Rf}$$

$$\begin{array}{l}\text{For each destructor } g,\\ \quad \text{for each reduction } g(M_1, \ldots, M_n) \rightarrow M \text{ in def}(g),\\ \qquad \text{attacker}(M_1) \wedge \ldots \wedge \text{attacker}(M_n) \Rightarrow \text{attacker}(M)\end{array} \tag{Rg}$$

$$\text{message}(x, y) \wedge \text{attacker}(x) \Rightarrow \text{attacker}(y) \tag{Rl}$$

$$\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{message}(x, y) \tag{Rs}$$

The rule (Init) represents the initial knowledge of the attacker. The rule (Rn) means that the attacker can generate an unbounded number of new names. The rules (Rf) and (Rg) mean that the attacker can apply all operations to all terms it has, (Rf) for constructors, (Rg) for destructors. Rule (Rl) means that the attacker can listen on all the channels it has, and (Rs) that it can send all the messages it has on all the channels it has.

Rules for the protocol. The translation $[\![P]\!]\rho h$ of a process P is a set of rules, where the environment ρ is a sequence of mappings $x \mapsto p$ and $a \mapsto p$ from variables and names to patterns, and h is a sequence of facts of the form

message(p, p'), and begin(p, ρ). The empty sequence is denoted by $\emptyset$, the concatenation of a fact F to the sequence h is denoted by $h \wedge F$. The concatenation of a mapping $u \mapsto p$ to ρ is denoted by $\rho[u \mapsto p]$, where u is a name or a variable. When M is a term, $\rho(M)$ is defined by considering ρ as a substitution: $\rho(u) = p$ when $u \mapsto p$ is in ρ, and $\rho(f(M_1, \ldots, M_n)) = f(\rho(M_1), \ldots, \rho(M_n))$ where f is a constructor. When V is a set of variables, $\rho(V)$ is a sequence of patterns containing p when $x \mapsto p$ is in ρ and $x \in V$, in the order of the sequence ρ.

$$
\begin{aligned}
&[\![0]\!]\rho h = \emptyset\\
&[\![P \mid Q]\!]\rho h = [\![P]\!]\rho h \cup [\![Q]\!]\rho h\\
&[\![!P]\!]\rho h = [\![P]\!](\rho[i \mapsto i])h \text{ where } i \text{ is a new variable (session identifier)}\\
&[\![(\nu a)P]\!]\rho h = [\![P]\!](\rho[a \mapsto a[\rho(V_o), \rho(V_s)]])h\\
&[\![M(x).P]\!]\rho h = [\![P]\!](\rho[x \mapsto x])(h \wedge \mathrm{message}(\rho(M), x))\\
&[\![\overline{M}\langle N\rangle.P]\!]\rho h = [\![P]\!]\rho h \cup \{h \Rightarrow \mathrm{message}(\rho(M), \rho(N))\}\\
&[\![\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho h = \cup\{[\![P]\!]((\sigma\rho)[x \mapsto \sigma' p'])(\sigma h)\\
&\quad \mid g(p'_1, \ldots, p'_n) \to p' \text{ is in } \mathrm{def}(g) \text{ and } (\sigma, \sigma') \text{ is a most general pair of}\\
&\quad \text{substitutions such that } \sigma\rho(M_1) = \sigma' p'_1, \ldots, \sigma\rho(M_n) = \sigma' p'_n\} \cup [\![Q]\!]\rho h\\
&[\![\mathtt{begin}(M).P]\!]\rho h = [\![P]\!]\rho(h \wedge \mathrm{begin}(\rho(M), \rho_{|V_o \cup V_s}))\\
&[\![\mathtt{end}(M).P]\!]\rho h = [\![P]\!]\rho h \cup \{h \Rightarrow \mathrm{end}(\rho(M), \rho(V_s))\}
\end{aligned}
$$

The translation of a process is a set of Horn clauses that enable us to prove that it sends certain messages or executes certain events. The sequence h keeps track of begin events that have happened and of messages received by the process, since these may trigger other messages. Replication inserts a new session identifier i in the environment ρ. The idea is that this session identifier takes a different value for each copy of the replicated process. Replication is otherwise ignored, because all Horn clauses are applicable arbitrarily many times.

For restriction, we replace the restricted name a in question with the pattern $a[\rho(V_o), \rho(V_s)]$. This pattern contains the previous inputs in $\rho(V_o)$ and session identifiers in $\rho(V_s)$.

To make the intuition on session identifiers more precise, we can consider instrumented processes in which the replication is annotated with an inequality $i \geq n$, indicating the session identifier i and the values it can take. The process $!^{i \geq n}P$ then represents copies of P for each $i \geq n$ in parallel, that is, $P\{n/i\} \mid P\{n+1/i\} \mid P\{n+2/i\} \mid \ldots$. The semantic rule $!P \to P \mid !P$ then becomes $!^{i \geq n}P \to P\{n/i\} \mid !^{i \geq n+1}P$. In the process, we replace the names by their corresponding patterns. Since the pattern contains the session identifiers, and these session identifiers get different values for different copies of the process, different names are mapped to different patterns, even when they are created by the same restriction. Such an instrumentation is formalized and used in the proofs.

The translation of a `begin` event adds an hypothesis to h, meaning that P can only be executed if the `begin` event has been executed first. This is similar to the translation of a `begin` event into a test in Theorem 1. The begin fact

has the part of the environment ρ containing variables as second parameter. This parameter will be useful for verifying injective agreement. In subsequent manipulations of this environment (in particular renamings), only the image is modified, the variables in the domain of ρ are left unchanged. (They are precisely used to remember names of variables in the original process.)

The translation of an `end` event adds a clause, meaning that the `end` event is triggered when all conditions in h are true. We can again notice similarity with the translation of an output, as suggested by Theorem 1. The second parameter of the end fact is the sequence of session identifiers of the session in which the `end` event is executed. This parameter is used for verifying injective agreement.

The other rules are as in [1]. The sequence h is extended in the translation of an input, with the input in question. The translation of an output adds a clause, meaning that the output is triggered when all conditions in h are true. The translation of a destructor application is the union of the clauses for the cases where the destructor succeeds (with an appropriate substitution) and where the destructor fails.

This translation of the protocol into Horn clauses introduces several approximations. We consider that the *else* clause of a destructor can always be executed. The actions are considered as implicitly replicated, since the rules can be executed any number of times. The exception to this point is the creation of new names: since session identifiers appear in patterns, the created name is precisely related to the session that creates it, so name creation cannot be unduly repeated inside the same session.

If $c \in S$, we replace all occurrences of $\text{message}(c[], M)$ by $\text{attacker}(M)$ in the rules. Indeed, both facts are equivalent by the rules (Rl) and (Rs).

Summary and results. Let $\rho = \{a \mapsto a[] \mid a \in fn(P_0)\}$. The rule base for P_0 is:

$$B_{P_0,S} = [\![P_0]\!]\rho\emptyset \cup \{\text{attacker}(a[]) \mid a \in S\} \cup \{(\text{Rn}), (\text{Rf}), (\text{Rg}), (\text{Rl}), (\text{Rs})\}$$

The rules for the process P of the end of Sect. 2 are rules for the attacker, plus:

$$\text{attacker}(pk(sk_A[])) \qquad \text{attacker}(pk(sk_B[]))$$

$$\text{attacker}(x\_pk_B) \wedge \text{begin}(x\_pk_B, \{\underline{i_A} \mapsto i_A, \underline{x\_pk_B} \mapsto x\_pk_B\})$$
$$\Rightarrow \text{attacker}(pk(sk_A[]))$$

$$\text{attacker}(x\_pk_B) \wedge \text{begin}(x\_pk_B, \{\underline{i_A} \mapsto i_A, \underline{x\_pk_B} \mapsto x\_pk_B\}) \wedge \text{attacker}(x\_b)$$
$$\Rightarrow \text{attacker}(sign((pk(sk_A[]), x\_pk_B, x\_b), sk_A[]))$$

$$\text{attacker}(x\_pk_A) \Rightarrow \text{attacker}(b[x\_pk_A, i_B])$$

$$\text{attacker}(x\_pk_A) \wedge \text{attacker}(sign((x\_pk_A, pk(sk_B[]), b[x\_pk_A, i_B]), sk_A[]))$$
$$\Rightarrow \text{end}(pk(sk_B[]), i_B)$$

The first two rules correspond to the outputs in P; they mean that the adversary has the public keys of the participants. The third and fourth rules correspond to the outputs in P_A. For example, the fourth rule means that if the attacker has x_pk_B and x_b and the event $\texttt{begin}(x\_pk_B)$ is allowed, then the attacker

can get $sign((pk(sk_A[]), x\_pk_B, x\_b), sk_A[])$, because P_A sends this message after receiving x_pk_B and x_b and executing the `begin` event. The variables in the domain of the environment in the begin fact are underlined to mark that they are not renamed when manipulating rules. The fifth rule corresponds to the output in P_B and the last one to the `end` event in P_B.

The following theorems give sufficient conditions for proving authenticity.

Theorem 2. *Assume that, for any B_b, if* $\mathrm{end}(T, s)$ *is derivable from $B_{P_0,S} \cup B_b$, then* $\mathrm{begin}(T, \rho) \in B_b$ *for some ρ. Then P_0 satisfies non-injective agreement with respect to S-adversaries.*

Intuitively, the condition means that $\mathrm{end}(T, s)$ cannot be derived without having $\mathrm{begin}(T, \rho)$ for some ρ in the hypotheses. Then the theorem says that $\mathtt{end}(M)$ cannot be executed without having first executed $\mathtt{begin}(M)$.

Theorem 3. *Let* $\mathrm{getsession}(\rho)$ *be the sequence of terms contained at the occurrence o in $\rho(x)$ for some x and o, assuming that o corresponds to the occurrence of s in a subterm of $\rho(x)$ of the form $a[\ldots, s, \ldots]$ where the restriction (νa) is under the last replication above* `end` *in P_0. Otherwise,* $\mathrm{getsession}(\rho)$ *is undefined.*

Assume that, for any B_b, if $\mathrm{end}(T, s)$ *is derivable from $B_{P_0,S} \cup B_b$, then* $\mathrm{begin}(T, \rho) \in B_b$ *with* $\mathrm{getsession}(\rho) = s$. *Then P_0 satisfies injective agreement with respect to S-adversaries.*[3]

The intuition behind this result is as follows. Assume that we add the session identifier s of the `end` event (i_B in the example), to the parameter M of the events `begin` and `end`. Then non-injective agreement on (M, s) implies injective agreement. Indeed, only one `end` event can be executed for each value of s. Therefore, $\mathtt{end}(M)$ executed n times corresponds to $\mathtt{end}(M, s_1), \ldots \mathtt{end}(M, s_n)$ executed for n distinct values $s_1, \ldots, s_n$, so $\mathtt{begin}(M, s_1), \ldots, \mathtt{begin}(M, s_n)$ have been executed, so $\mathtt{begin}(M)$ has been executed at least n times.

We can easily add s to $\mathtt{end}(M)$, but for $\mathtt{begin}(M)$, the session identifier s is in general not in scope. This session identifier is going to appear in data received by the process, which are collected in environment ρ. So we add the environment ρ as parameter to the begin fact. The function getsession is going to extract the session identifier s from ρ, so that intuitively we prove non-injective agreement between $\mathtt{begin}(M_1, \mathrm{getsession}(\rho))$ and $\mathtt{end}(M_2, s)$, to obtain injective agreement.

If we delay the emission of `begin` inside the same session, we can only decrease the number of emissions of `begin`:

Lemma 1. *Let $P_1 = C[\mathtt{begin}(M).D[P]]$, where no replication occurs in $D[]$ above the hole $[]$. Let $P_2 = C[D[\mathtt{begin}(M).P]]$. If P_2 satisfies injective agreement with respect to S-adversaries, then so does P_1.*

[3] Actually, we can prove, with the same hypotheses, that P_0 satisfies *recent* injective agreement, in the sense that the runtimes of the sessions that emit the `begin` and `end` events overlap. This point is not detailed here because of space constraints.

4.3 Solving Algorithm

To determine whether a fact is derivable from $B_{P_0,S} \cup B_b$, we adapt the solving algorithm of [6]. Very roughly, the algorithm combines rules by resolution, starting from a set of rules B_0. From two rules $R = H \Rightarrow C$ and $R' = H' \Rightarrow C'$ it infers $\sigma(H \cup (H' - F_0)) \Rightarrow \sigma C'$, where C and $F_0 \in H'$ are unifiable, σ is the most general unifier of C and F_0, $\mathrm{sel}(R) = \emptyset$, $F_0 \in \mathrm{sel}(R')$, and sel is a selection function: $\mathrm{sel}(H \to C) = \emptyset$ if $H \subseteq$ *Unselectable*, $\mathrm{sel}(H \to C) = \{F_0\}, F_0 \in H -$ *Unselectable* of maximum size otherwise, with *Unselectable* $= \{\mathrm{attacker}(x), x \text{ variable}\}$. So facts of the form $\mathrm{attacker}(x)$ are never selected. (In some rare cases, more sophisticated selection functions can be necessary to obtain termination, as explained in [6, extended version]. For simplicity, we do not consider this case here.) When no new rules can be added, we define $\mathrm{saturate}(B_0)$ as the set of obtained rules R such that $\mathrm{sel}(R) = \emptyset$. As shown in [6], a fact F is derivable from B_0 if and only if it is derivable from $\mathrm{saturate}(B_0)$. Intuitively, saturate transforms a set of rules into an equivalent but simpler one: the hypotheses of the resulting rules are all of the form $\mathrm{attacker}(x)$.

We conjecture that, for protocols using only public channels, public-key cryptography with atomic keys, shared-key cryptography and hash functions, the algorithm using the function sel above terminates when each encryption, signature, and hash computation can be immediately distinguished from others in the protocol, for example by a tag: for instance, when we want to encrypt m under k, we add the tag c_0 to m, so that the encryption becomes $\mathit{sencrypt}((c_0, m), k)$ where the tag c_0 is a different constant for each encryption in the protocol. This condition is easy to realize by adding tags, and it is also a good protocol design: the participants use the tags to identify the messages unambiguously. We are currently proving this point with A. Podelski.

In the case of authenticity, we modify the selection function such that it *never* selects facts of the form $\mathrm{begin}(p, \rho)$. (Then saturate returns rules containing hypotheses $\mathrm{attacker}(x)$ and $\mathrm{begin}(p, \rho)$.) An easy extension of [6] shows that:

Theorem 4. *F is derivable from $B_0 \cup B_b$ if and only if it is derivable from* $\mathrm{saturate}(B_0) \cup B_b$.

Intuitively, if an hypothesis $\mathrm{begin}(p, \rho)$ appears in all rules of $\mathrm{saturate}(B_0)$ that prove $\mathrm{end}(p, s)$, then this hypothesis is necessary to derive $\mathrm{end}(p, s)$. We obtain the following condition to prove non-injective agreement:

Theorem 5. *Assume that, for each $R \in \mathrm{saturate}(B_{P_0,S})$ such that $R = H \Rightarrow \mathrm{end}(p, s)$, we have $\mathrm{begin}(p, \rho) \in H$ for some ρ. Then P_0 satisfies non-injective agreement with respect to S-adversaries.*

Proof sketch. If $\mathrm{end}(T, s)$ is derivable from $B_{P_0,S} \cup B_b$, then it is derivable from $\mathrm{saturate}(B_{P_0,S}) \cup B_b$, by Theorem 4. The last rule R of this derivation has $\mathrm{end}(p, s')$ in its conclusion $R = H \Rightarrow \mathrm{end}(p, s')$, so by hypothesis, $\mathrm{begin}(p, \rho') \in H$. So $\mathrm{begin}(T, \rho)$ must be derivable from $\mathrm{saturate}(B_{P_0,S}) \cup B_b$. Since the only rules that prove $\mathrm{begin}(T, \rho)$ are the closed facts of B_b, $\mathrm{begin}(T, \rho) \in B_b$. So we have non-injective agreement by Theorem 2. □

To prove injective agreement thanks to Theorem 3 and Lemma 1, we have to find the right position of $\texttt{begin}(M)$, since Lemma 1 allows to move it, and the value of the getsession function. We proceed as follows. First, we replace ρ in the hypothesis $\text{begin}(p, \rho)$ of each rule, by an environment that contains all variables defined at the output that creates the rule and that are not defined under a replication under $\texttt{begin}(M)$, instead of only the variables defined at the $\texttt{begin}(M)$ event. That is, $B'_{P_0,S}$ is the set of rules defined as $B_{P_0,S}$ except that

$$\begin{aligned}
&[\![\overline{M}\langle N\rangle.P]\!]\rho h = [\![P]\!]\rho h \cup \{\ h\{\rho_{|V_o \cup V_s}/X\} \Rightarrow \text{message}(\rho(M), \rho(N))\ \}\\
&[\![!P]\!]\rho h = [\![P]\!](\rho[i \mapsto i])(h\{\rho_{|V_o \cup V_s}/X\})\\
&[\![\texttt{begin}(M).P]\!]\rho h = [\![P]\!]\rho(h \wedge \text{begin}(\rho(M), X))
\end{aligned}$$

where X is a special variable. Let $\text{occ}(s, p)$ be the set of occurrences of the sequence of session identifiers s in the pattern p that appear in a subpattern of the form $a[\ldots, s, \ldots]$, where the restriction (νa) is under the last replication above $\texttt{end}$ in P_0. Let $\text{occ}(s, \rho) = \{(x, o) \mid x \in dom(\rho), o \in \text{occ}(s, \rho(x))\}$ and $\text{occ}(B) = \cap_{H,p,s \text{ such that } H \Rightarrow \text{end}(p,s) \in B} \cup_{\rho \text{ such that } \text{begin}(p,\rho) \in H} \text{occ}(s, \rho)$.

Theorem 6. *If* $\text{occ}(\text{saturate}(B'_{P_0,S})) \neq \emptyset$*, then* P_0 *satisfies injective agreement with respect to* S*-adversaries.*

Proof sketch. Consider $(x, o) \in \text{occ}(\text{saturate}(B'_{P_0,S}))$. If $\texttt{begin}(M)$ is not under the binder of x in P_0, we move it just under the binder of x. Let P_1 be the obtained process. The rules of $B_{P_1,S}$ are the rules of $B'_{P_0,S}$, except that the hypotheses $\text{begin}(p, \rho)$ in which ρ does not contain x are removed, and in the other hypotheses $\text{begin}(p, \rho)$, ρ is restricted to contain only variables defined above x, including x. The rules in $\text{saturate}(B_{P_1,S})$ are transformed in the same way, so we still have $(x, o) \in \text{occ}(\text{saturate}(B_{P_1,S}))$. So for all $R \in \text{saturate}(B_{P_1,S})$, if $R = H \Rightarrow \text{end}(p, s)$ then $\text{begin}(p, \rho) \in H$ with $o \in \text{occ}(s, \rho(x))$. Similarly to the proof of Theorem 5, if $\text{end}(T, s)$ is derivable from $B_{P_1,S} \cup B_b$, then $\text{begin}(T, \rho) \in B_b$ with $o \in \text{occ}(s, \rho(x))$. The function $\text{getsession}(\rho)$ is chosen to return the term at occurrence o in $\rho(x)$, so $\text{getsession}(\rho) = s$. By Theorem 3, P_1 satisfies injective agreement with respect to S-adversaries, and by Lemma 1 so does P_0. □

In the example of Sect. 2, the first rule for P_A is unchanged, and the second one becomes:

$$\begin{aligned}
&\text{attacker}(x_pk_B) \wedge \text{attacker}(x_b) \wedge \text{begin}(x_pk_B, \{\underline{i_A} \mapsto i_A, \underline{x_pk_B} \mapsto x_pk_B,\\
&\quad \underline{x_b} \mapsto x_b\}) \Rightarrow \text{attacker}(sign((pk(sk_A[]), x_pk_B, x_b), sk_A[]))
\end{aligned}$$

instead of having $\text{begin}(x\_pk_B, \{\underline{i_A} \mapsto i_A, \underline{x\_pk_B} \mapsto x\_pk_B\})$ in all rules for P_A. The solving algorithm then yields

$$\begin{aligned}
&\text{begin}(pk(sk_B[]), \{\underline{i_A} \mapsto i_A, \underline{x_pk_B} \mapsto pk(sk_B[]), \underline{x_b} \mapsto b[pk(sk_A[]), i_B]\})\\
&\quad \Rightarrow \text{end}(pk(sk_B[]), i_B)
\end{aligned}$$

Since x_b contains i_B, we move `begin` under the input that binds x_b. The process P_A then becomes:

$$P_{A2}(sk_A, pk_A) = c(x\_pk_B).\overline{c}\langle pk_A\rangle.c(x\_b).\texttt{begin}(x\_pk_B)\ldots$$

We take getsession$(\rho) = i$ if $\rho(x\_b) = b[p, i]$, and getsession is undefined otherwise, so that getsession$(\rho) = i_B$ for the values of the variables obtained above. Thanks to Theorem 3, we prove that the modified process containing P_{A2} satisfies injective agreement with respect to $\{c\}$-adversaries, and by Lemma 1, we obtain the same result for P.

5 Experimental Results and Conclusion

Our experimental results are summarized in Fig. 4. The tests have been performed on a Pentium III, 1Ghz processor, with a verifier implemented in Ocaml 3.04. The protocols are given with a reference to the paper in which they appeared. The second column indicates the number of rules that represent the protocol. The third one gives the mean time needed to check one specification for each protocol.

	# rules	Time (ms)
Needham-Schroeder public key [20]	21	25
Needham-Schroeder public key corrected [18]	21	16
Woo-Lam public key [26]	18	4
Woo-Lam public key corrected [28]	18	6
Woo-Lam shared key [14]	16	6
Woo-Lam shared key corrected [14]	15	5
Simpler Yahalom [9], unidirectional	10	29
Simpler Yahalom [9], bidirectional	13	101
Otway-Rees [21]	12	62
Simpler Otway-Rees [3]	12	10
Main mode of Skeme [17]	19	67

Fig. 4. Experimental results

The first six protocols (Needham-Schroeder public key and Woo-Lam protocols) are authentication protocols. For them, we have tested non-injective and injective agreement on the name of the participants, and on all atomic data.

For the Woo and Lam shared-key protocol, our tool proves the correctness of the corrected versions of [5] and [14]; it finds an attack on the flawed version of [14]. (The messages received or sent by A do not depend on the host A wants to talk to, so A may start a session with the adversary C, and the adversary can reuse the messages of this session to talk to B in A's name.) We can easily see that the versions of [26] and [3, Example 6.2] are also subject to this attack, even if our tool does not terminate on them. The only difference between the protocol of [14] and that of [26] is that [14] adds tags to distinguish different encryption

sites. Using tags is certainly a good design practice, and our analyzer terminates when tags are added. Our tool finds the attack of [10, bottom of page 52] on the versions of [3, end of Example 3.2] and [28].

The last five protocols exchange a session key, so we have tested agreement on the names of the participants, and agreement on both the participants and the session key.

For all protocols, except the non-termination for two versions of Woo-Lam shared key, we found known attacks against flawed protocols, and proved the correctness of the corrected versions. No false attack was found. These results show that our fully automatic technique yields an efficient verifier for authenticity.

Acknowledgments. We would like to thank Martín Abadi for stimulating discussions on this work, Jérôme Feret and Andrew Gordon for helpful comments on a draft of this paper.

References

1. M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. In *29th Annual ACM Symposium on Principles of Programming Languages (POPL 2002)*, pages 33–44, Portland, Oregon, Jan. 2002. ACM Press.
2. M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *28th Annual ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, London, United Kingdom, Jan. 2001. ACM Press.
3. M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, Jan. 1996.
4. R. Amadio and S. Prasad. The game of the name in cryptographic tables. In P. S. Thiagarajan and R. Yap, editors, *Advances in Computing Science - ASIAN'99*, volume 1742 of *LNCS*, pages 15–27, Phuket, Thailand, Dec. 1999. Springer Verlag.
5. R. Anderson and R. Needham. Programming Satan's Computer. In J. van Leeuven, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 426–440. Springer Verlag, 1995.
6. B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society. Extended version available at `http://www.di.ens.fr/~blanchet/longcsfw14.ps.gz`.
7. P. Broadfoot, G. Lowe, and B. Roscoe. Automating Data Independence. In *6th European Symposium on Research in Computer Security (ESORICS 2000)*, volume 1895 of *LNCS*, pages 175–190, Toulouse, France, Oct. 2000. Springer Verlag.
8. P. J. Broadfoot and A. W. Roscoe. Internalising agents in CSP protocol models. In *Workshop on Issues in the Theory of Security (WITS'02)*, Portland, Oregon, Jan. 2002.
9. M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.
10. J. Clark and J. Jacob. A Survey of Authentication Protocol Literature: Version1.0. Technical report, University of York, Department of Computer Science, Nov. 1997.
11. E. Cohen. TAPS: A First-Order Verifier for Cryptographic Protocols. In *13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 144–158, Cambridge, England, July 2000.

12. V. Cortier, J. Millen, and H. Rueß. Proving secrecy is easy enough. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 97–108, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.
13. M. Debbabi, M. Mejri, N. Tawbi, and I. Yahmadi. A New Algorithm for the Automatic Verification of Authentication Protocols: From Specifications to Flaws and Attack Scenarios. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, Rutgers University, New Jersey, Sept. 1997.
14. A. Gordon and A. Jeffrey. Authenticity by Typing for Security Protocols. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 145–159, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.
15. A. Gordon and A. Jeffrey. Types and Effects for Asymmetric Cryptographic Protocols. In *15th IEEE Computer Security Foundations Workshop (CSFW-15)*, Cape Breton, Nova Scotia, Canada, June 2002. IEEE Computer Society.
16. J. Heather and S. Schneider. Towards automatic verification of authentication protocols on an unbounded network. In *13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 132–143, Cambridge, England, July 2000.
17. H. Krawczyk. SKEME: A Versatile Secure Key Exchange Mechanism for Internet. In *Proceedings of the Internet Society Symposium on Network and Distributed Systems Security*, Feb. 1996.
18. G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *LNCS*, pages 147–166. Springer Verlag, 1996.
19. G. Lowe. A Hierarchy of Authentication Specifications. In *Proceedings of the 10th Computer Security Foundations Workshop (CSFW '97)*, Rockport, Massachusetts, June 1997. IEEE Computer Society.
20. R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM*, 21(12):993–999, Dec. 1978.
21. D. Otway and O. Rees. Efficient and Timely Mutual Authentication. *Operating Systems Review*, 21(1):8–10, 1987.
22. L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.
23. A. W. Roscoe and P. J. Broadfoot. Proving Security Protocols with Model Checkers by Data Independence Techniques. *Journal of Computer Security*, 7(2, 3):147–190, 1999.
24. D. X. Song. Athena: a New Efficient Automatic Checker for Security Protocol Analysis. In *12th IEEE Computer Security Foundation Workshop (CSFW-12)*, Mordano, Italy, June 1999.
25. C. Weidenbach. Towards an Automatic Analysis of Security Protocols in First-Order Logic. In H. Ganzinger, editor, *16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 314–328, Trento, Italy, July 1999. Springer Verlag.
26. T. Y. C. Woo and S. S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, Jan. 1992.
27. T. Y. C. Woo and S. S. Lam. A Semantic Model for Authentication Protocols. In *Proceedings IEEE Symposium on Research in Security and Privacy*, pages 178–194, Oakland, California, May 1993.
28. T. Y. C. Woo and S. S. Lam. Authentication for distributed systems. In D. Denning and P. Denning, editors, *Internet Besieged: Countering Cyberspace Scofflaws*. ACM Press and Addison-Wesley, Oct. 1997.

Security Typings by Abstract Interpretation

Mirko Zanotti

Dipartimento di Informatica
Università di Verona
Strada Le Grazie 15 a Ca' Vignal 2
I-37134 Verona, Italy

LIX — Laboratoire d'Informatique
École Polytechnique
91128 Palaiseau cedex, France

mirko.zanotti@tiscali.it

Abstract. Starting from a denotational semantics of imperative programs, the standard collecting semantics is defined as specifying the strongest program property. By successive Galois connection based abstractions, a type system that certifies secure information flows is designed as an abstract semantics approximating the collecting semantics of imperative programs. Security abstract analysis embodies the secure flow conditions of Denning's lattice model (see [6]), giving rise to a type system which is similar to, but strictly more expressive than, the one proposed by Volpano et al. in [10]. This shows that types and type systems for control flow analysis can be viewed as abstract interpretations.

1 Introduction

Starting from years '70, security became an important issue of information systems, initially with the precise scope to grant data right access. In this direction, security is viewed as an *internal* property of an information system: only flows of information involving known entities inside the system are considered. Unauthorized accesses to the information system from external agents are not considered here. This kind of analysis can be applied to imperative programs, that are information systems where variables are known internal entities, each of which has a security level. The aim of the analysis is the control of information flows among variables, to ensure that the content of sensitive (or secret) data will not be stored in wrong variables (i.e variables that does not have an adequate security level).

A first approach to secure flows of information was made by Lampson in [9], while Bell and LaPadula with [1] formalized the first mathematical model where the security property of information flows can be proved. The model was extended by Denning in [6] (see also [5] and [7]) with its *lattice model*, which represents the first important result about security in the field of program certification. In the lattice model, a security policy on information flows is a lattice $\langle SC, \leq \rangle$, where SC is a finite set of security classes with partial order $\leq$.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 360–375, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Every variable x used inside a program has a security class, $\underline{x}$. It is supposed that the security class of each variable is statically determined, and that it can't be modified at run time.

> **Secure Information Flow:** *If x_1 and x_2 are program's variables and there is an information flow from x_1 to x_2, the flow is allowable if and only if $\underline{x_1} \leq \underline{x_2}$.*

In this way, every programming construct (e.g. assignment, loop, etc.) can be coupled with a purely syntactic certification condition. This condition refers to the security classes of the construct's variables in order to verify the soundness of the information flow generated when the construct will be executed.

Denning's lattice model still remains a reference work for programs' secure information flows analysis, but its weakness is the lack of a soundness proof with respect to the standard semantics of imperative languages. This proof was obtained by Volpano-Smith-Irvine in [10], by means of a type system that certifies secure information flows for a generic deterministic imperative language. The type system guarantees that well-typed programs in a procedural language satisfy a *noninterference* security property. With all program inputs and outputs classified at various security levels, the property basically states that a program output, classified at some level, can never change as a result of modifying only inputs classified at higher levels. Intuitively, this means that the program does not "leak" sensitive data.

The characterization of the lattice model with a type system allows to clearly devise the security policies from the algorithms that realize them inside the programs. In this way, one can use automatic type inference techniques in order to verify flows of information for real programs. On the other side, type systems have some general handicaps. First of all, the definition of a type system is mainly intuitive. This affects demonstrations of related properties, because any change in a type system requires a new proof that old properties apply for the new system. Hence, type systems generally don't allow high extendibility of obtained results.

This paper proposes a systematic method to derive a type system that checks for secure flows inside the kernel of a generic imperative programming language, guaranteeing the noninterference security property in Volpano et al. style. The reference work for the derivation of type systems by means of abstract interpretation is the general framework proposed by Patrick Cousot in [2]. Here various type systems for eager untyped lambda-calculus are systematically derived by abstract interpretation starting from the standard collecting semantics of functional languages. By a first abstraction, a new sound type collecting semantics is derived in compositional fix-point form. Then, by successive (semi-dual) Galois connection based abstractions, type systems and/or type inference algorithms are designed as abstract semantics or abstract interpreters approximating the type collecting semantics. This leads to a hierarchy of type systems, which is part of the lattice of abstract interpretations of the untyped lambda-calculus. The basic idea for this paper is to adapt [2] to the standard semantics of imperative languages in order to perform secure flow analysis. Due to the two level denotational semantics of imperative languages, two separate analyses will

be performed, one for expressions and one for programs. Both analyses embody Denning's secure flows conditions, leading to the definition of a type system similar to Volpano-Smith-Irvine's one. The analysis of expressions and the analysis of programs have a common outline, that can be summarized with two sequential steps: a first Galois connection is established to define security types, then this connection is extended to derive security typings as abstractions of the standard collecting semantics. Moreover, secure flow abstract analysis requires to modify the framework proposed by Cousot in order to satisfy the following two needs:

- an imperative language is considered, whose standard denotational semantics is stratified for expressions and programs;
- security types must be derived, whose concretizations are memory location properties rather than value properties.

The derivation process, built for the kernel of an imperative programming language, defines a type system that grants secure information flows satisfying Denning's certification conditions. The result of the abstract analysis is strictly more expressive than the original type system proposed by Volpano-Smith-Irvine in [10]. In fact, the type system for imperative programs systematically derived by abstract interpretation allows to know the security class of *each* variable assigned within a secure program. Moreover, both type systems for expressions and programs proposed in [10], will be obtained by approximation of the corresponding type systems systematically derived by abstract interpretation. The great benefit in the use of abstract interpretation framework to derive security typings is the high modularity and extendibility achieved: the derivation process is systematic and soundness properties are granted by construction. So, if new commands are added to the imperative kernel language considered, only part of the analysis will be modified, and soundness of the new abstraction will be still obtained by construction.

2 Preliminaries

In the following $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$ denotes a complete lattice C, with ordering $\leq$, *lub* $\vee$, *glb* $\wedge$, greatest element (top) $\top$, and least element (bottom) $\bot$. The downward closure of $S \subseteq C$ is defined as $\downarrow S \stackrel{\text{def}}{=} \{x \in C \mid \exists y \in S.\ x \leq y\}$. $\downarrow x$ is a shorthand for $\downarrow \{x\}$, while the upward closure $\uparrow$ is dually defined. We consider here Galois insertion/connection based abstract interpretation [3]. If A and C are Posets, and $\alpha : C \xrightarrow{\text{m}} A$ and $\gamma : A \xrightarrow{\text{m}} C$ are monotone functions such that $\forall x \in C.\ x \leq \gamma(\alpha(x))$ and $\forall x \in A.\ \alpha(\gamma(x)) \leq x$, then the quadruple (α, C, A, γ) is called a *Galois connection* (GC for short) between C and A. The concrete and abstract domains, C and A, are supposed to be complete lattices and are related by abstraction and concretization maps forming a GC (α, C, A, γ). If in addition $\forall a \in A.\ \alpha(\gamma(a)) = a$, then we call (α, C, A, γ) a Galois insertion (GI) of A in C. When (α, C, A, γ) is a GI then each value of the abstract domain A is useful in representing C, because all the elements of A represent distinct members of C, being γ 1-1. Given two complete lattices A and C, a GC (α, C, A, γ) is established if and only if the concretization function γ is co-additive, $\forall a_1, a_2 \in A.\ \gamma(a_1 \wedge a_2) = \gamma(a_1) \wedge \gamma(a_2)$, if and only if the

abstraction function α is additive, $\forall c_1, c_2 \in C.\, \alpha(c_1 \vee c_2) = \alpha(c_1) \vee \alpha(c_2)$. Any GC may be lifted to a GI identifying in an equivalence class those values of the abstract domain with the same concretization. This process is known as *reduction* of the abstract domain. An abstract domain A corresponds to a complete meet sub-semilattice of C (i.e., $\wedge$ is its *glb*), but, in general, it does not correspond to a complete sublattice of C, since the *lub* induced by A in C— namely $\gamma(\alpha(\vee Y))$ — might be different from that in C (i.e. $\vee Y$). Indeed, the two *lub*$'s$ coincide whenever $\gamma(\alpha(C))$ is a complete sublattice of C, which holds iff γ is additive. In this case we say that A is *disjunctive*. The lattice of abstract interpretations of C [4], is denoted $\langle \mathcal{L}_C, \sqsubseteq, \sqcap, \sqcup, C, \{\top\}\rangle$, with C being the bottom abstract domain (most concrete abstraction) and $\{\top\}$ being the top abstract domain (most abstract abstraction) of C. In this case $A \sqsubseteq B$ iff $B \subseteq A$ as Moore families of C.

3 Systematic Construction of Security Types

The aim of the paper is the systematic derivation of a type system that certifies secure information flows for imperative programs. Due to the high complexity in modern imperative languages, a restricted subset of these languages will be taken in this paper. Computability theory ensures that an imperative language made up of assignment, iteration and composition of commands is Turing-complete. Hence the language considered for the derivation process will be the following:

$$\begin{array}{rlll} \textit{Expressions} & e \in \mathbb{E} & e ::= v \mid x \mid e \pm e & v \in \mathbb{V}, x \in \mathbb{X} \\ \textit{Programs} & c \in \mathbb{C} & c ::= x := e \mid \textit{while } e \textit{ do } \{c\} \mid c; c & \end{array}$$

where $\mathbb{V}$ is the set of *values* (domain of the calculus) and $\mathbb{X}$ is the set of *variables*. In what follows, $\mathbb{R} \stackrel{\text{def}}{=} \mathbb{X} \mapsto \mathbb{V}$[1] is the set of *environments*, $\mathbb{S} \stackrel{\text{def}}{=} \mathbb{R} \mapsto \mathbb{R}$ is the *semantic domain* (of programs), $\mathbb{S}^{\mathrm{E}} \stackrel{\text{def}}{=} \mathbb{R} \mapsto \mathbb{V}$ is the *semantic domain of expressions*. The denotational semantics of the language is stratified for expressions and programs (see [11]). Semantics of expressions is defined by $\mathbf{S}^{\mathrm{E}}[\![\cdot]\!] : \mathbb{E} \longrightarrow \mathbb{S}^{\mathrm{E}}$ such that:

$$\begin{aligned} & v \mapsto \lambda R \in \mathbb{R}.\, v \\ & x \mapsto \lambda R \in \mathbb{R}.\, R(x) \\ & e_1 \pm e_2 \mapsto \lambda R \in \mathbb{R}.\, (\mathbf{S}^{\mathrm{E}}[\![e_1]\!](R) \pm \mathbf{S}^{\mathrm{E}}[\![e_2]\!](R)) \end{aligned}$$

[1] **Notation**
The symbol $\stackrel{\text{def}}{=}$ denotes a definition.
The symbol $\mapsto$ between two sets D and C denotes the set of functions mapping elements of D in elements of C.
Given a generic $R \in \mathbb{R}$, the writing $R[x \leftarrow e]$ will denote the environment that assigns the expression e to the variable x, coinciding with R for any variable $y \neq x$.
The function $Cond : \mathbb{V} \times \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$ will denote the test "less than zero", i.e. $Cond(v, R_1, R_2) = R_1$ if $v < 0$, R_2 otherwise.

The semantics of programs is defined by $\mathbf{S}[\![\cdot]\!] : \mathbb{C} \longrightarrow \mathbb{S}$ such that:

$$x := e \mapsto \lambda R \in \mathbb{R}.\, R[x \leftarrow \mathbf{S}^{\mathrm{E}}[\![e]\!](R)]$$
$$c_1; c_2 \mapsto \lambda R \in \mathbb{R}.\, \mathbf{S}[\![c_2]\!](\mathbf{S}[\![c_1]\!](R))$$
$$\mathit{while}\ e\ \mathit{do}\ \{c\} \mapsto \lambda R \in \mathbb{R}.\, \mathit{Cond}(\mathbf{S}^{\mathrm{E}}[\![e]\!](R),\, \mathbf{S}[\![c; \mathit{while}\ e\ \mathit{do}\ \{c\}]\!](R),\, R)$$

A *semantic property* is a set of semantics. The separation of the denotational semantics for expressions and programs leads to the definition of two kind of semantic properties: $\mathbb{P}^{\mathrm{E}} \stackrel{\mathrm{def}}{=} \wp(\mathbb{S}^{\mathrm{E}})$ for expressions, and $\mathbb{P} \stackrel{\mathrm{def}}{=} \wp(\mathbb{S})$ for programs. The *standard collecting semantics* is the function that maps every object in its stronger semantic property, precisely in the set having as unique element the standard semantics of that object. Still, for imperative languages there can be distinguished two standard collecting semantics, $\mathbf{C}^{\mathrm{E}}[\![\cdot]\!] : \mathbb{E} \longrightarrow \mathbb{P}^{\mathrm{E}}$ such that $e \mapsto \{\mathbf{S}^{\mathrm{E}}[\![e]\!]\}$ for expressions, and $\mathbf{C}[\![\cdot]\!] : \mathbb{C} \longrightarrow \mathbb{P}$ such that $c \mapsto \{\mathbf{S}[\![c]\!]\}$ for programs.

3.1 Security Analysis as Abstract Interpretation

In order to define the desired security typing, we suppose the availability of n *security classes* $\mathbb{M}^{\mathrm{V}} \stackrel{\mathrm{def}}{=} \{s_1 \mathit{var}, \ldots, s_n \mathit{var}\}$, with n fixed "a priori". Each security class can be statically assigned to the variables of an imperative program, by means of type declarations. The security class of each variable corresponds to the "sensitivity level" of data that will be assigned to the variable itself. These hypotheses fully agree with Denning's analysis (see [5],[6] and [7]).

Given the set of available security classes, security abstract analysis is well-founded if the following conditions are satisfied:

- for each security class $s_i \mathit{var} \in \mathbb{M}^{\mathrm{V}}$ there is a set of locations $\mathcal{L}_i$ having an accessibility level corresponding to the sensitivity level intended for $s_i \mathit{var}$. This is specified by the concretization function $\gamma_1^{\mathrm{V}} : \mathbb{M}^{\mathrm{V}} \longrightarrow \wp(\mathbb{L})$ such that $s_i \mathit{var} \mapsto \mathcal{L} = \{l \in \mathbb{L} \mid l \text{ has accessibility level } s_i\}$, where $\mathbb{L}$ is the set of available locations. We suppose that the concretizations of different security classes are disjoint;
- data assigned to a variable with security class $s_i \mathit{var}$ have to be stored in a location belonging to $\mathcal{L}_i$. Given a type environment $H \in \mathbb{H} \stackrel{\mathrm{def}}{=} \mathbb{X} \mapsto \mathbb{M}^{\mathrm{V}}$ and an environment $R \in \mathbb{R} \stackrel{\mathrm{def}}{=} \mathbb{X} \mapsto \mathbb{V}$, the following condition must be satisfied:

$$\forall x \in \mathbb{X}.\, R(x) \neq \bot \implies \mathit{loc}(x) \in \gamma_1^{\mathrm{V}}(H(x))\ .$$

  where $\mathit{loc} : \mathbb{X} \longrightarrow \mathbb{L}$ is the one-to-one function mapping every variable in its memory location. In this way, every type environment H can be associated to the set of environments which are coherent with H:

$$\gamma_2^{\mathrm{V}}(H) \stackrel{\mathrm{def}}{=} \{R \in \mathbb{R} \mid \forall x \in \mathbb{X}.\, R(x) \neq \bot \implies \mathit{loc}(x) \in \gamma_1^{\mathrm{V}}(H(x))\}\ .$$

Type systems are understood as abstract compositional semantics approximating the standard collecting semantics of imperative languages. The idea is to build

a Galois-connection between security types and semantic properties, such that the abstraction of the semantic property P defines the best possible type for a program having property P. Due to the two-levels definition of the imperative languages semantics, the typing derivation must be performed in two steps, firstly for expressions and then for commands.

3.2 Security Analysis of Expressions

The aim of the security analysis of expressions is the systematic derivation of a security typing for expressions by means of abstract interpretation. This typing should associate every expression e with the set of locations where e can be stored, in accordance with a security policy that defines the allowable information flows within a program. We suppose that a set $\mathbb{M}^{\mathrm{V}}$ containing n security classes is available. $\mathbb{M}^{\mathrm{V}}$ will be used to statically assign every variable with a confidentiality level, which must be corresponded by an accessibility level of memory locations in which variables are mapped.

Security Types. By hypothesis, each security class $s_i\,var$ denotes a set of locations $\mathcal{L}_i$, where data having sensitivity level s_i can be stored. The allowable information flows are determined when it is specified where expressions can be stored. In what follows, the writing $s_I\,exp$ will denote the security type of expressions that can be stored inside any location of $\mathcal{L}_i$ with $i \in I$. For example, given two security types $s_{\{1\}}\,exp$ and $s_{\{1,2\}}\,exp$, the information flows from $s_{\{1\}}\,exp$ to $s_{\{1,2\}}\,exp$ are allowed, but not conversely.

A set $\mathbb{M}^{\mathrm{E}}$ containing security types is *well-formed* if and only if $\mathbb{M}^{\mathrm{E}}$ is closed by intersection and $s_{\{1,\ldots,n\}}\,exp = \top^{\mathrm{E}} \in \mathbb{M}^{\mathrm{E}}$. The security type $\top^{\mathrm{E}} = s_{\{1,\ldots,n\}}\,exp$ must always belong to $\mathbb{M}^{\mathrm{E}}$, in order to denote data that can be stored in any memory location, such as numeric constants. Well-formed sets of security types are supposed to be partially ordered by the following relation $\leq^{\mathrm{E}}$:

$$\forall s_{I_1}\,exp, s_{I_2}\,exp \in \mathbb{M}^{\mathrm{E}}.\ s_{I_1}\,exp \leq^{\mathrm{E}} s_{I_2}\,exp \iff \gamma_1^{\mathrm{E}}\,(s_{I_1}\,exp) \subseteq \gamma_1^{\mathrm{E}}\,(s_{I_2}\,exp)$$

where $\gamma_1^{\mathrm{E}} : \mathbb{M}^{\mathrm{E}} \longrightarrow \wp(\mathbb{L})$ maps every security type in the set of locations where expressions with that type can be stored: $\gamma_1^{\mathrm{E}}\,(s_I\,exp) = \bigcup_{i \in I} \gamma_1^{\mathrm{V}}(s_i\,var)$.

The smaller is the security type, the higher is the secrecy of expressions having that type, the lower is the possibility to know the content of variables having that type.

Proposition 1. $\langle\, \wp(\mathbb{L}), \subseteq, \emptyset, \mathbb{L}, \cap, \cup \,\rangle \underset{\alpha_1^{\mathrm{E}}}{\overset{\gamma_1^{\mathrm{E}}}{\leftrightarrows}} \langle\, \mathbb{M}^{\mathrm{E}}, \leq^{\mathrm{E}}, \bot^{\mathrm{E}}, \top^{\mathrm{E}}, \bigwedge^{\mathrm{E}}, \bigvee^{\mathrm{E}} \,\rangle$ *is a Galois connection.*

Security Typing. Now it is possible to define the Galois connection that approximates semantic properties of expressions, $\mathbb{P}^{\mathrm{E}}$, with security typings for expressions, $\mathbb{T}^{\mathrm{E}} \stackrel{\mathrm{def}}{=} \mathbb{H} \mapsto \mathbb{M}^{\mathrm{E}}$. $\mathbb{T}^{\mathrm{E}}$ is assumed to be partially ordered by the relation $\ddot{\leq}^{\mathrm{E}}$:

$$\forall \theta_1, \theta_2 \in \mathbb{T}^{\mathrm{E}}.\ \theta_1 \ddot{\leq}^{\mathrm{E}} \theta_2 \iff \forall H \in \mathbb{H}.\, \theta_1(H) \leq^{\mathrm{E}} \theta_2(H)\ .$$

The desired Galois connection is established with the definition of the abstraction function $\alpha^{\mathrm{E}} : \mathbb{P}^{\mathrm{E}} \longrightarrow \mathbb{T}^{\mathrm{E}}$ such that:

$$P \mapsto \lambda H \in \mathbb{H}.\, \alpha_1^{\mathrm{E}} \left(\{ loc(x) \in \mathbb{L} \mid \exists \sigma \in P : \forall R \in \gamma_2^{\mathrm{V}}(H). R(x) \neq \bot \Longrightarrow \sigma R \neq \bot \} \right).$$

In order to understand this definition it should be noted that the set:

$$\{ loc(x) \in \mathbb{L} \mid \exists \sigma \in \mathbf{C}^{\mathrm{E}}[\![e]\!] : \forall R \in \gamma_2^{\mathrm{V}}(H).\ R(x) \neq \bot \implies \sigma R \neq \bot \}$$

identifies the locations associated to the variables that are inside the expression e. Moreover, semantics inside $\mathbf{C}^{\mathrm{E}}[\![e]\!]$ are evaluated only with environments R coherent with the type environment H (i.e. $R \in \gamma_2^{\mathrm{V}}(H)$). This ensures, for every location associated to a variable of e, that the accessibility level of that location agrees with the security class of the corresponding variable.

Proposition 2. $\langle \mathbb{P}^{\mathrm{E}}, \subseteq, \emptyset, \mathbb{S}^{\mathrm{E}}, \cap, \cup \rangle \underset{\alpha^{\mathrm{E}}}{\overset{\gamma^{\mathrm{E}}}{\leftrightarrows}} \langle \mathbb{T}^{\mathrm{E}}, \ddot{\leq}^{\mathrm{E}}, \ddot{\bot}^{\mathrm{E}}, \ddot{\top}^{\mathrm{E}}, \ddot{\bigwedge}^{\mathrm{E}}, \ddot{\bigvee}^{\mathrm{E}} \rangle$ *is a Galois connection.*

The security typing for expressions is obtained by upper approximation of the abstraction α_1^{E} applied to the standard collecting semantics of expressions $\mathbf{C}^{\mathrm{E}}[\![\cdot]\!]$. Structural induction is used to perform the derivation process on expressions, providing at the same time a soundness proof for the typing.

Theorem 1. *The derived security typing for expressions* $\mathbf{T}^{\mathrm{E}}[\![\cdot]\!]$ *is well defined and sound, i.e. for every expression* $e \in \mathbb{E}$ *the following property holds:*

$$\alpha^{\mathrm{E}} \left(\mathbf{C}[\![e]\!] \right) \ddot{\leq}^{\mathrm{E}} \mathbf{T}^{\mathrm{E}}[\![e]\!] \ .$$

Finally, the security typing for expressions systematically derived by abstract interpretation, $\mathbf{T}^{\mathrm{E}}[\![\cdot]\!] : \mathbb{E} \longrightarrow \mathbb{T}^{\mathrm{E}}$, can be defined as follows:

$$\begin{aligned} n &\mapsto \lambda H \in \mathbb{H}.\, \top^{\mathrm{E}} \\ x &\mapsto \lambda H \in \mathbb{H}.\, \alpha_1^{\mathrm{E}}(\mathcal{L}_i) \text{ if } H(x) = s_i var \\ e_1 \pm e_2 &\mapsto \ddot{\bigwedge}_{i=1,2}^{\mathrm{E}} \mathbf{T}^{\mathrm{E}}[\![e_i]\!] \end{aligned}$$

where α_1^{E} is the security policy.

Example 1. Let's assume the availability of three security classes, so the set $\mathbb{M}^{\mathrm{V}} = \{ s_1 var,\ s_2 var,\ s_3 var \}$ is given. Each security class must correspond to a set of locations having adequate accessibility level, as it is suggested by the concretization function $\gamma_1^{\mathrm{V}} : \mathbb{M}^{\mathrm{V}} \longrightarrow \wp(\mathbb{L})$ such that $s_i var \mapsto \mathcal{L}$ for $i = 1, 2, 3$. Sets $\mathcal{L}_i$ are supposed to be disjoint, $\forall i \neq j.\ \mathcal{L}_i \cap \mathcal{L}_j = \emptyset$, each of which having different accessibility level. Now the security classes must be correlated among them in order to define the lattice of expressions' security types. Indeed, the security classes $s_1 var$, $s_2 var$, $s_3 var$ identify three different locations accessibility levels, but only the security policy defines the allowable information flows among these levels.

For example, one could allow only information flows from locations of $\mathcal{L}_i$ to locations of $\mathcal{L}_j$ with $j \leq i$. A security policy must be defined, such that in the security types lattice one can distinguish at least three points:

- a security type for expressions containing informations coming from locations of $\mathcal{L}_1$. These expressions have to be stored only inside locations belonging to the set $\mathcal{L}_1$ itself;
- a security type for expressions containing informations coming from locations of $\mathcal{L}_2$ that does not refer to data contained inside locations of $\mathcal{L}_1$. These expressions have to be stored inside any location of $\mathcal{L}_2$;
- a security type for expressions containing informations coming only from locations of $\mathcal{L}_3$. These expressions can be stored inside any location.

The lattice of security types for expressions that allows all the required information flows, is $\mathbb{M}^{\mathrm{E}} = \{s_{\{1\}}\,exp,\ s_{\{1,2\}}\,exp,\ s_{\{1,2,3\}}\,exp\}$. By recalling the definition of order on $\mathbb{M}^{\mathrm{E}}$, the lattice of security types for expressions corresponds to a chain, which satisfies exactly the request that only information flows from variables having security type $s_i\,var$ to variables having security type $s_j\,var$ with $j \leq i$ are allowed. In order to check how the derived security typing for expressions works, the variables' declaration **var** $x_1 : s_1\,var,\ x_2 : s_2\,var,\ x_3 : s_3\,var$ is considered, and the security type of expression $e = x_1 \pm x_2 \pm x_3$ will be determined. Variables' declaration defines the type environment $H = \{x_i \mapsto s_i\,var\ i = 1, 2, 3\}$, which determine the desired instance of the security typing $\mathbf{T}^{\mathrm{E}}[\![e]\!]$:

$$\begin{aligned}
\mathbf{T}^{\mathrm{E}}[\![x_1 \pm x_2 \pm x_3]\!](H) &= \mathbf{T}^{\mathrm{E}}[\![x_1]\!](H) \bigwedge^{\mathrm{E}} \mathbf{T}^{\mathrm{E}}[\![x_2]\!](H) \bigwedge^{\mathrm{E}} \mathbf{T}^{\mathrm{E}}[\![x_3]\!](H) \\
&= \alpha_1^{\mathrm{E}}(\mathcal{L}_1) \bigwedge^{\mathrm{E}} \alpha_1^{\mathrm{E}}(\mathcal{L}_2) \bigwedge^{\mathrm{E}} \alpha_1^{\mathrm{E}}(\mathcal{L}_3) \\
&= s_{\{1\}}\,exp \bigwedge^{\mathrm{E}} s_{\{1,2\}}\,exp \bigwedge^{\mathrm{E}} s_{\{1,2,3\}}\,exp \\
&= s_{\{1\}}\,exp\ .
\end{aligned}$$

The obtained security type highlights that the value of expression e can be stored only inside the locations of $\mathcal{L}_1$. This fully agrees with the hypothesis that $s_{\{1\}}\,exp$ is the most restrictive security type.

3.3 Security Analysis of Programs

The aim of the programs' security analysis is the systematic derivation of a security typing for imperative programs, by means of abstract interpretation. A program P should admit a security type if and only if every information flow within P is in accordance with the security policy established in the analysis of expressions. In terms of abstract interpretation, the Galois connection $(\mathbb{P},\ \alpha,\ \mathbb{T},\ \gamma)$ approximating semantic properties $\mathbb{P}$ with security typings $\mathbb{T}$ must be defined.

Security Types. The basic idea is that the security type of a program P should collect the security classes of variables receiving a secure assignment in P. The notion of allowable information flow within a program is defined by structural induction on programs:

- the assignment $x := e$ defines an explicit information flow from expression e to variable x. The information flow is allowable if and only if the security type of e is lower than the security type of x;

- the iteration *while e do* $\{c\}$ defines, besides the information flows of command c, an implicit information flow from the expression e to every variable assigned within c. The information flow is allowable if and only if the security type of e is lower than the security type of every variable assigned in c;
- the composition of commands $c_1; c_2$ joins the information flows of c_1 and c_2. The information flow is allowable if and only if the flows of c_1 and c_2 are allowable.

The lattice of security types is defined as $\mathbb{M} \stackrel{\text{def}}{=} \{s_I\, cmd, \top\}$ with $I \in \wp(\{1, \ldots, n\})$. A first concretization function $\gamma_1 : \mathbb{M} \longrightarrow \wp(\mathbb{R})$ must be defined, mapping each security type in the set of environments that can be generated executing a safe command of that type. For the sake of clarity, $S_I(c)$ will denote the property "c is a safe command with security type $s_I\, cmd$":

$$
\begin{aligned}
&S_I(x := e) \iff \exists i \in I : loc(x) \in \gamma_1^{\mathrm{V}}(s_i\, var), \mathbf{T}^{\mathrm{E}}[\![x]\!] \ddot{\leq}^{\mathrm{E}} \mathbf{T}^{\mathrm{E}}[\![e]\!] \\
&S_I(\textit{while } e \textit{ do } \{c\}) \iff \mathbf{S}[\![c]\!]R \in \gamma_1(s_I\, cmd), \forall x \in av(c).\, \mathbf{T}^{\mathrm{E}}[\![x]\!] \ddot{\leq}^{\mathrm{E}} \mathbf{T}^{\mathrm{E}}[\![e]\!] \\
&S_I(c_1; c_2) \iff \mathbf{S}[\![c_1]\!]R \in \gamma_1(s_I\, cmd),\ \mathbf{S}[\![c_2]\!]R \in \gamma_1(s_I\, cmd)
\end{aligned}
$$

The concretization function γ_1 can be inductively defined as follows:

$$
\begin{aligned}
&\gamma_1(s_\emptyset\, cmd) = \emptyset \\
&\gamma_1(s_I\, cmd) = \{\mathbf{S}[\![c]\!]R \mid c \in \mathbb{C} \text{ and } S_I(c) \text{ and } R \in \mathbb{R}\} \\
&\gamma_1(\top) = \mathbb{R}
\end{aligned}
$$

where $av(c)$ is the set of variables assigned inside c. The meaning of this function could still seem quite vague, so some further examples will be given here. Let's suppose the availability of two variables x_1 and x_2, such that x_1 has a security class lower (i.e. more secure) than x_2. Any environment $R \in \mathbb{R}$ generated executing the command "$x := 1$" will belong to $\gamma_1\big(s_{\{1\}}\, cmd\big)$, because its information flow is allowable and by hypothesis $loc(x) \in \gamma_1^{\mathrm{V}}(s_1\, var)$ holds. On the contrary, any environment generated executing the command "*while* x *do* $\{y := 0\}$" will not belong to the concretization of a security type. In fact, the security type of y is greater than the security type of x, hence an incorrect implicit information flow could be generated by that command. The set $\mathbb{M}$ of programs' security types is assumed to be partially ordered by the following relation:

$$
\forall s_{I_1}\, cmd, s_{I_2}\, cmd \in \mathbb{M}.\ s_{I_1}\, cmd \leq s_{I_2}\, cmd \iff \gamma_1(s_{I_1}\, cmd) \subseteq \gamma_1(s_{I_2}\, cmd)\ .
$$

Hence, if the writings $\bot$ and $\top$ identify bottom and top elements of lattice $\mathbb{M}$, the following equations apply: $\bot = s_\emptyset\, cmd$, $\top = s_{\{1,\ldots,n\}}\, cmd$. It must be proved that the function γ_1 induces a Galois connection between the set of programs' security types and the environments' properties.

Proposition 3. $\langle \wp(\mathbb{R}), \subseteq, \emptyset, \mathbb{R}, \cup, \cap \rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \mathbb{M}, \leq, \bot, \top, \bigvee, \bigwedge \rangle$ *is a Galois connection.*

Security Typing. Finally it is possible to define the Galois connection that approximates semantic properties with security typings for imperative programs, $\mathbb{T} \stackrel{\text{def}}{=} \mathbb{H} \mapsto \mathbb{M}$. $\mathbb{T}$ is assumed to be partially ordered by the following relation $\ddot{\leq}$:

$$\forall \theta_1, \theta_2 \in \mathbb{T}.\ \theta_1 \ddot{\leq} \theta_2 \iff \forall H \in \mathbb{H}.\ \theta_1(H) \leq \theta_2(H)\ .$$

The desired Galois connection is established with the definition of the abstraction function $\alpha : \mathbb{P} \longrightarrow \mathbb{T}$, such that:

$$\alpha(P) = \lambda H \in \mathbb{H}.\, \alpha_1\left(\{\sigma(R) \,|\, \sigma \in P \text{ and } R \in \gamma_2^{\mathrm{V}}(H) \text{ and } \sigma(R) \in \gamma_2^{\mathrm{V}}(H)\}\right).$$

Proposition 4. $\langle \mathbb{P}, \subseteq, \emptyset, \mathbb{S}, \cup, \cap \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{T}, \ddot{\leq}, \ddot{\bot}, \ddot{\top}, \ddot{\bigvee}, \ddot{\bigwedge} \rangle$ *is a Galois connection.*

The security typing for programs is obtained by upper approximation of the abstraction α applied to the standard collecting semantics of programs $\mathbf{C}[\![\cdot]\!]$. Structural induction on programs is used to perform the derivation, providing at the same time a soundness proof for the derived security typing.

Theorem 2. *The derived security typing for programs* $\mathbf{T}[\![\cdot]\!]$ *is well defined and sound, i.e. for every command* $c \in \mathbb{C}$ *the following property holds:*

$$\alpha(\mathbf{C}[\![c]\!]) \ddot{\leq} \mathbf{T}[\![c]\!]\ .$$

The obtained type system $\mathbf{T}[\![\cdot]\!] : \mathbb{C} \longrightarrow \mathbb{T}$, certifying secure information flows for imperative programs, can be written as follows:

$$x := e \mapsto \lambda H \in \mathbb{H}.\ \begin{cases} s_{\{i\}}\, cmd & \text{if} \quad H(x) = s_i\, var,\ \mathbf{T}^{\mathrm{E}}[\![x]\!] \ddot{\leq}^{\mathrm{E}} \mathbf{T}^{\mathrm{E}}[\![e]\!] \\ \top & \text{otherwise} \end{cases}$$

$$\textit{while } e \textit{ do } \{c\} \mapsto \lambda H \in \mathbb{H}.\ \begin{cases} \mathbf{T}[\![c]\!](H) & \text{if} \quad \forall x \in av(c).\, \mathbf{T}^{\mathrm{E}}[\![x]\!] \ddot{\leq}^{\mathrm{E}} \mathbf{T}^{\mathrm{E}}[\![e]\!] \\ \top & \text{otherwise} \end{cases}$$

$$c_1; c_2 \mapsto \ddot{\bigvee}_{i=1,2} \mathbf{T}[\![c_i]\!]$$

The problem of this typing is that the function $av(\cdot)$ must be used in order to verify the soundness of implicit information flows. The following lemma allows the specification of a recursive side condition for the loop.

Lemma 1. *Given* n *security classes* $s_1 var, \ldots, s_n var$, *the following property holds:* $\forall e \in \mathbb{E}. \forall c \in \mathbb{C}. \forall I \in \wp(\{1, \ldots, n\}). \forall H \in \mathbb{H}.$

$$\mathbf{T}[\![c]\!](H) = s_I\, cmd,\ s_I\, exp \leq^{\mathrm{E}} \mathbf{T}^{\mathrm{E}}[\![e]\!](H) \implies \forall x \in av(c).\, \mathbf{T}^{\mathrm{E}}[\![x]\!] \ddot{\leq}^{\mathrm{E}} \mathbf{T}^{\mathrm{E}}[\![e]\!]\ .$$

Finally, the following typing $\mathbf{T}[\![\cdot]\!]$ is assumed as the definitive security typing for imperative programs derived by abstract interpretation:

$$x := e \mapsto \lambda H \in \mathbb{H}. \begin{cases} s_{\{i\}}\, cmd & \text{if} \quad H(x) = s_i\, var,\ \mathbf{T}^{\mathrm{E}}[\![x]\!] \ddot{\leq}^{\mathrm{E}} \mathbf{T}^{\mathrm{E}}[\![e]\!] \\ \top & \text{otherwise} \end{cases}$$

$$\mathit{while}\ e\ \mathit{do}\ \{c\} \mapsto \lambda H \in \mathbb{H}. \begin{cases} \mathbf{T}[\![c]\!](H) & \text{if} \quad \begin{array}{l} \mathbf{T}[\![c]\!](H) = s_I\, cmd, \\ s_I\, exp \leq^{\mathrm{E}} \mathbf{T}^{\mathrm{E}}[\![e]\!](H) \end{array} \\ \top & \text{otherwise} \end{cases}$$

$$c_1; c_2 \mapsto \ddot{\bigvee_{i=1,2}} \mathbf{T}[\![c_i]\!]$$

Example 2. Let's suppose that two security classes $s_1\, var$ and $s_2\, var$ are available, with $s_2\, var$ identifying more important data than $s_1\, var$. The lattice of security types for expressions is $\mathbb{M}^{\mathrm{E}} = \{s_{\{2\}}\, exp,\ s_{\{1,2\}}\, exp\}$. This implies that every variable declared with security class $s_1\, var$ (and every numeric constant) will have security type $s_{\{1,2\}}\, exp$, while every variable declared with security class $s_2\, var$ will have security type $s_{\{2\}}\, exp$. The security type of the following program P will be determined:

```
var x:s2var, y:s1var
y:=0;
while x do {x := x − 1; y := y + 1}
```

Variables' declaration defines the type environment $H = \{x \mapsto s_2\, var,\ y \mapsto s_1\, var\}$ which determine the desired instance of the security typing $\mathbf{T}[\![P]\!]$:

$$\begin{aligned} \mathbf{T}[\![P]\!](H) &= \mathbf{T}[\![y := 0]\!](H) \bigvee \mathbf{T}[\![\mathit{while}\ x\ \mathit{do}\ \{x := x - 1;\ y := y + 1\}]\!](H) \\ &= s_{\{1\}}\, cmd \bigvee \begin{cases} \mathbf{T}[\![c]\!](H) & \text{if} \quad \mathbf{T}[\![c]\!] = s_I\, cmd,\ s_I\, exp \leq^{\mathrm{E}} \mathbf{T}^{\mathrm{E}}[\![x]\!](H) \\ \top & \text{otherwise} \end{cases} \\ &\quad \text{where } c = \{x := x - 1;\ y := y + 1\}\ . \end{aligned}$$

The security type of the loop's body must be derived in order to determine the soundness of the implicit information flows inside P:

$$\begin{aligned} \mathbf{T}[\![c]\!](H) &= \mathbf{T}[\![x := x - 1]\!](H) \bigvee \mathbf{T}[\![y := y + 1]\!](H) \\ &= s_{\{2\}}\, cmd \bigvee s_{\{1\}}\, cmd \\ &= s_{\{1,2\}}\, cmd\ . \end{aligned}$$

Now the security type of P can be derived:

$$\begin{aligned} \mathbf{T}[\![P]\!](H) &= s_{\{1\}}\, cmd \bigvee \begin{cases} s_{\{1,2\}}\, cmd & \text{if} \quad s_{\{1,2\}}\, exp \leq^{\mathrm{E}} s_{\{2\}}\, exp \\ \top & \text{otherwise} \end{cases} \\ &= \top\ . \end{aligned}$$

The typing derived by abstract interpretation does not certify the program P, detecting an implicit information flow that could be unsafe.

4 Correlation with the Analysis of Volpano et al.

In this section we prove the type system proposed by Volpano et al. in [10] is an approximation of the type system derived by abstract interpretation. In order to do that, the first one will be obtained from the latter by means of an abstraction function defining a Galois connection. Some remarks must previously be done. By convention, in abstract interpretation, ordering relation denotes the stronger property. In the security types lattice, the represented property is data accessibility; hence the relation $s_{I_1} exp \leq^{\mathrm{E}} s_{I_2} exp$ means that the type $s_{I_1} exp$ is less accessible (therefore it is more secure) than the type $s_{I_2} exp$. On the contrary, in accordance with Denning's lattice model of information flows, in Volpano analysis the ordering relation on security types of expressions denotes more accessibility. In other terms, the relation $s_{i_1} \leq s_{i_2}$ defined by Volpano et al. means that the content of variables having security type s_{i_1} is more accessible than the content of variables having security type s_{i_2}. In spite of this, both analyses have the same meaning:

- a numeric constant has higher accessibility level (less restrictive security type) and his value can be transferred in every memory location;
- the composition of two expressions has accessibility level lower than the accessibility level of both composing expressions, hence a composed expression is always more secure than its components.

The lattice of security types for expressions defined by Volpano et al., $\mathbb{M}^{\mathrm{V_E}} = \{s_1, \ldots, s_n\}$ (where n is the number of security classes), can be obtained from the similar lattice defined for abstract analysis, $\mathbb{M}^{\mathrm{E}}$, keeping its join-irreducible elements. In this way, for every index $i = 1 \ldots n$, the type $s_i \in \mathbb{M}^{\mathrm{V_E}}$ will correspond to the smallest type $s_I exp$ such that $i \in I$, i.e. to the security type $\alpha_1^{\mathrm{E}}(\mathcal{L}_i)$. This is accomplished with the definition of the concretization function $\gamma_1^{\mathrm{V_E}} : \mathbb{M}^{\mathrm{V_E}} \longrightarrow \mathbb{M}^{\mathrm{E}}$ such that:

$$\gamma_1^{\mathrm{V_E}}(s_i) = \alpha_1^{\mathrm{E}}(\mathcal{L}_i), \quad \gamma_1^{\mathrm{V_E}}\left(\bot^{\mathrm{V_E}}\right) = \top^{\mathrm{E}}$$

with ordering relation $\forall s_{i_1}, s_{i_2} \in \mathbb{M}^{\mathrm{V_E}}.\ s_{i_1} \geq^{\mathrm{V_E}} s_{i_2} \iff \gamma_1^{\mathrm{V_E}}(s_{i_1}) \leq^{\mathrm{E}} \gamma_1^{\mathrm{V_E}}(s_{i_2})$.

The function $\gamma^{\mathrm{V_E}}$is not injective, so two distinct types s_i, s_j could match the same type $s_I exp$. This simply means that the security policy defined for expressions associates to types s_i and s_j the same meaning, i.e. the two types represents the same security class. Being the lattice $\mathbb{M}^{\mathrm{E}}$ complete, the same property holds for $\mathbb{M}^{\mathrm{V_E}}$, which is derived from the $\mathbb{M}^{\mathrm{E}}$ by keeping its join-irreducible elements.

Proposition 5. $\langle \mathbb{M}^{\mathrm{E}}, \leq^{\mathrm{E}}, \bot^{\mathrm{E}}, \top^{\mathrm{E}}, \bigwedge^{\mathrm{E}}, \bigvee^{\mathrm{E}} \rangle \underset{\alpha_1^{\mathrm{V_E}}}{\overset{\gamma_1^{\mathrm{V_E}}}{\leftrightarrows}} \langle \mathbb{M}^{\mathrm{V_E}}, \geq^{\mathrm{V_E}}, \top^{\mathrm{V_E}}, \bot^{\mathrm{V_E}}, \bigvee^{\mathrm{V_E}}, \bigwedge^{\mathrm{V_E}} \rangle$ *is a Galois connection.*

First of all, the security typing *for expressions* proposed by Volpano et al., $\mathbb{T}^{\mathrm{V_E}} \stackrel{\mathrm{def}}{=} \mathbb{H} \mapsto \mathbb{M}^{\mathrm{V_E}}$, will be obtained from the similar type system derived by abstract interpretation. In order to do that, the Galois connection $\langle \alpha_1^{\mathrm{V_E}}, \gamma_1^{\mathrm{V_E}} \rangle$ is extended with the definition of the concretization function $\gamma^{\mathrm{V_E}} : \mathbb{T}^{\mathrm{V_E}} \longrightarrow \mathbb{T}^{\mathrm{E}}$ such that:

$$\gamma^{\mathrm{V_E}}(\theta) = \lambda H \in \mathbb{H}.\, \gamma_1^{\mathrm{V_E}}(\theta(H))$$

with ordering relation $\forall \theta_1, \theta_2 \in \mathbb{T}^{\mathrm{V_E}}.\ \theta_1 \ddot{\geq}^{\mathrm{V_E}} \theta_2 \iff \gamma^{\mathrm{V_E}}(\theta_1) \ddot{\leq}^{\mathrm{E}} \gamma^{\mathrm{V_E}}(\theta_2)$.

Proposition 6. $\langle \mathbb{T}^{\mathrm{E}}, \ddot{\leq}^{\mathrm{E}}, \ddot{\bot}^{\mathrm{E}}, \ddot{\top}^{\mathrm{E}}, \ddot{\bigwedge}^{\mathrm{E}}, \ddot{\bigvee}^{\mathrm{E}} \rangle \underset{\alpha^{\mathrm{V_E}}}{\overset{\gamma^{\mathrm{V_E}}}{\leftrightarrows}} \langle \mathbb{T}^{\mathrm{V_E}}, \ddot{\geq}^{\mathrm{V_E}}, \ddot{\top}^{\mathrm{V_E}}, \ddot{\bot}^{\mathrm{V_E}}, \ddot{\bigvee}^{\mathrm{V_E}}, \ddot{\bigwedge}^{\mathrm{V_E}} \rangle$ *is a Galois connection.*

By applying the abstraction function $\alpha^{\mathrm{V_E}}$ to $\mathbf{T}^{\mathrm{E}}[\![e]\!]$ for every expression $e \in \mathbb{E}$, the following security typing $\mathbf{V}^{\mathrm{E}}[\![\cdot]\!] : \mathbb{E} \longrightarrow \mathbb{T}^{\mathrm{V_E}}$ can be derived, which coincide with the security typing for expressions defined by Volpano et al.:

$$n \mapsto \lambda H \in \mathbb{H}.\, \bot^{\mathrm{V_E}}$$
$$x \mapsto \lambda H \in \mathbb{H}.\, s_i \text{if } H(x) = s_i\, var$$
$$e_1 \pm e_2 \mapsto \ddot{\bigvee}_{i=1,2}^{\mathrm{V_E}} \mathbf{V}^{\mathrm{E}}[\![e_i]\!]$$

Finally, we show that the security typing *for programs* proposed by Volpano et al. can be obtained from the similar typing derived by abstract interpretation. The set of security types for programs defined by Volpano et al. is $\mathbb{M}^{\mathrm{V}} = \{ s_i\, cmd \,|\, s_i \in \mathbb{M}^{\mathrm{V_E}} \}$. Roughly speaking, this states that if a program P has security type $s_i\, cmd$, then P is secure and inside it there are not assigned variables having security class lower than s_i. For the same program, the security type derived by abstract interpretation collects the security class of any variable assigned inside P. Hence the abstraction function $\alpha_1^{\mathrm{V}} : \mathbb{M} \longrightarrow \mathbb{M}^{\mathrm{V}}$ can be given as:

$$\alpha_1^{\mathrm{V}}(s_I\, cmd) = s_i\, cmd \qquad s_i = \bigwedge_{j \in I}^{\mathrm{V_E}} s_j$$

with partial order $\forall s_{i_1} cmd, s_{i_2} cmd \in \mathbb{M}^{\mathrm{V}}.\ s_{i_1} cmd \leq^{\mathrm{V}} s_{i_2} cmd \iff s_{i_1} \geq^{\mathrm{V_E}} s_{i_2}$.

Proposition 7. $\langle \mathbb{M}, \leq, \bot, \top, \bigwedge, \bigvee \rangle \underset{\alpha_1^{\mathrm{V}}}{\overset{\gamma_1^{\mathrm{V}}}{\leftrightarrows}} \langle \mathbb{M}^{\mathrm{V}}, \leq^{\mathrm{V}}, \bot^{\mathrm{V}}, \top^{\mathrm{V}}, \bigwedge^{\mathrm{V}}, \bigvee^{\mathrm{V}} \rangle$ *is a Galois connection.*

By extending the latter Galois connection $\langle \alpha_1^{\mathrm{V}}, \gamma_1^{\mathrm{V}} \rangle$, a new security typing for program $\mathbb{T}^{\mathrm{V}} \stackrel{\mathrm{def}}{=} \mathbb{H} \mapsto \mathbb{M}^{\mathrm{V}}$ can be derived. The desired extension can be defined by the concretization function $\gamma^{\mathrm{V}} : \mathbb{T} \longrightarrow \mathbb{T}^{\mathrm{V}}$ such that:

$$\gamma^{\mathrm{V}}(\theta) = \lambda H \in \mathbb{H}.\, \gamma_1^{\mathrm{V}}(\theta(H))$$

with partial order $\forall \theta_1, \theta_2 \in \mathbb{T}^{\mathrm{V}}.\ \theta_1 \ddot{\leq}^{\mathrm{V}} \theta_2 \iff \gamma^{\mathrm{V}}(\theta_1) \ddot{\leq} \gamma^{\mathrm{V}}(\theta_2)$.

Proposition 8. $\langle \mathbb{T}, \ddot{\leq}, \ddot{\bot}, \ddot{\top}, \ddot{\bigwedge}, \ddot{\bigvee} \rangle \underset{\alpha^{\mathrm{V}}}{\overset{\gamma^{\mathrm{V}}}{\leftrightarrows}} \langle \mathbb{T}^{\mathrm{V}}, \ddot{\leq}^{\mathrm{V}}, \ddot{\bot}^{\mathrm{V}}, \ddot{\top}^{\mathrm{V}}, \ddot{\bigwedge}^{\mathrm{V}}, \ddot{\bigvee}^{\mathrm{V}} \rangle$ *is a Galois connection.*

By applying the abstraction function α^{V} to $\mathbf{T}[\![c]\!]$ for every command $c \in \mathbb{C}$, the following security typing $\mathbf{V}[\![\cdot]\!] : \mathbb{C} \longrightarrow \mathbb{T}^{\mathrm{V}}$ will be derived, which coincide with the security typing for programs defined by Volpano et al.:

$$x := e \mapsto \lambda H \in \mathbb{H}. \begin{cases} s_i\, cmd & \text{if} \quad \mathbf{V}^{\mathrm{E}}[\![x]\!] = s_i,\ \mathbf{V}^{\mathrm{E}}[\![e]\!] \ddot{\leq}^{\mathrm{V_E}} \mathbf{V}^{\mathrm{E}}[\![x]\!] \\ \top^{\mathrm{V}} & \text{otherwise} \end{cases}$$

$$\textit{while } e \textit{ do } \{c\} \mapsto \lambda H \in \mathbb{H}. \begin{cases} \mathbf{T}[\![c]\!](H) & \text{if} \quad \begin{array}{l} \mathbf{V}[\![c]\!](H) = s_i\, cmd, \\ \mathbf{V}^{\mathrm{E}}[\![e]\!](H) \ \leq^{\mathrm{V_E}}\ s_i\, exp \end{array} \\ \top^{\mathrm{V}} & \text{otherwise} \end{cases}$$

$$c_1; c_2 \mapsto \ddot{\bigvee}_{i=1,2}^{\mathrm{V}} \mathbf{V}[\![c_i]\!]$$

Example 3. Let's suppose that two security classes $s_1\, var$ and $s_2\, var$ are available, with $s_2\, var$ identifying more important data than $s_1\, var$. The lattice of security types for expressions is $\mathbb{M}^{\mathrm{E}} = \{s_{\{2\}}\, exp,\ s_{\{1,2\}}\, exp\}$. This implies that every variable declared with security class $s_1\, var$ (and every numeric constant) will have security type $s_{\{1,2\}}\, exp$, while every variable declared with security class $s_2\, var$ will have security type $s_{\{2\}}\, exp$. The problem is to know whether the following program P can be certified as secure or not:

```
var x:s1var, y:s2var
x:=2;  y:=0;
while x do {x := x − 1; y := y + 1}
```

Variables' declaration defines the type environment $H = \{x \mapsto s_2\, var,\ y \mapsto s_1\, var\}$ so the security types $\mathbf{T}[\![P]\!](H)$ and $\mathbf{V}[\![P]\!](H)$ can be determined. Both typings will agree that program P is secure, because all of its explicit and implicit information flows are allowable. The security type of P is $\mathbf{T}[\![P]\!](H) = s_{\{1,2\}}\, cmd$, where one can see that inside P variables with security class $s_1\, var$ and $s_2\, var$ are assigned. The security type of P that is obtained with the type system defined by Volpano et al. is $\mathbf{V}[\![P]\!](H) = s_2\, cmd$, where one can only know that inside P variables with security class $s_2\, var$ are assigned. This shows how the type system derived by abstract interpretation in this paper is more expressive than the original type system proposed in [10] by Volpano et al.

5 Conclusions

A type system certifying secure information flows has been systematically derived by abstract interpretation, representing a generalization of a similar model proposed by Volpano et al. in [10]. This latter is a (less expressive) particular case of the type system derived in this paper, as it has been proved in the previous section. Besides the improvements given by the analysis presented here, there are some general advantages typical of abstract interpretation framework:

- *systematic derivation* of the type system, whose soundness is a constructive characteristic. There is no need to "invent" the type system and prove its correctness;
- *extendibility* of the obtained results. If the language is enhanced with other commands, it suffices to add the security conditions of the command in the abstraction function in order to systematically re-derive the security typing;

– *refining* obtained results is very simple. A refinement of the abstract domain corresponds to a refinement of the security policy. Moreover, the reduced product theory can be used in order to unify different security policies in a unique policy that fully represents them (see [8]).

A further development of the security abstract analysis should extend the simple imperative language considered here, improving the derived type system in order to address real-languages needs (i.e. pointer analysis, multi-threading, etc.). Moreover, an implementation of the derived type system could be very useful. In fact, it may be the case that the additional accuracy of the abstract analysis w.r.t. [10] may also add an important overhead in practical cases. Finally, as a completely different issue, it could be interesting to know how security abstract analysis fits in the lattice of type analysis designed by P. Cousot in [2].

We think, like Volpano and Smith do, that "a truly secure programming language requires fundamental changes in language design, and an understanding of relationships between semantics and security". There's no way to achieve real secure information flows without forcing programmers to make use of security classes. Today's cryptographic algorithms can perhaps ensure privacy of communications, but these techniques are meaningless if applications that manage data do not guarantee stability and check for secure internal flows of information. Only by verifying programs with static and/or dynamic techniques (like type-based approach) more stable software will be achieved, leading to really secure information systems.

Acknowledgments. I'm very grateful to Roberto Giacobazzi, who gave me many suggestions about the organization of the paper, together with useful critical reviews. I also thank Radhia Cousot and the "semantics, proof and abstract interpretation" equipe at École Polytechnique in Paris, where I worked for six months during my Erasmus fellowship.
This work has been supported by the Italian MURST-COFIN-2000 Project "Abstract Interpretation, type systems and control-flow analysis".

References

1. D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical report, Air Force Systems Command, 1973.
2. P. Cousot. Types as abstract interpretations. In *Conference Record of the 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 316–331. ACM Press, New York, U.S.A., 1997.
3. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symp. on Principles of Programming Languages* (*POPL '77*), pages 238–252. ACM Press, New York, 1977.
4. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symp. on Principles of Programming Languages* (*POPL '79*), pages 269–282. ACM Press, New York, 1979.
5. D. E. Denning. *Secure Information Flow in Computer Systems.* PhD thesis, Purdue University, 1975.

6. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.
7. D. E. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
8. R. Giacobazzi and F. Ranzato. Refining and compressing abstract domains. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proc. of the 24th Internat. Colloq. on Automata, Languages and Programming* (*ICALP '97*), volume 1256 of *Lecture Notes in Computer Science*, pages 771–781. Springer-Verlag, Berlin, 1997.
9. B. W. Lampson. Dynamic protection structures. *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 27–38, 1969.
10. 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.
11. G. Winskel. *The Formal Semantics of Programming Languages: an Introduction*. Foundations of Computing Series. The MIT Press, 1993.

Static Confidentiality Enforcement for Distributed Programs*

Andrei Sabelfeld[1] and Heiko Mantel[2]

[1] Department of Computer Science
Cornell University, Ithaca, NY 14853, USA, andrei@cs.cornell.edu
[2] German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, mantel@dfki.de

Abstract. Preserving the confidentiality of data in a distributed system is an increasingly important problem of current security research. Distributed programming often involves message passing over a publicly observable medium, which opens up various opportunities for eavesdropping. Not only may the contents of messages sent on a *public* channel reveal confidential data, but merely observing the presence of a message on a channel for *encrypted* traffic may leak information. Another source of leaks is *blocking,* which may change the observable behavior of a process that attempts to receive on an empty channel.

In this article, we investigate the interplay between, on the one side, public, encrypted, and private (or hidden) channels of communication and, on the other side, blocking and nonblocking communication primitives for a simple multi-threaded language. We argue for timing-sensitive security and give a compositional timing-sensitive confidentiality specification. A key contribution of this article is a security-type system that statically enforces confidentiality. That the type system is not over-restrictive is exemplified by a typable distributed file-server program.

1 Introduction

Standard security infrastructure provides no satisfactory guarantees for preserving the *confidentiality* of data that is manipulated by computing systems. Consider a file-client program that accesses a distributed file server and manipulates confidential files via a publicly accessible network. In order to preserve the data's confidentiality, an array of security mechanisms is typically involved. First, an *access control* policy may enforce an authorized manipulation of files. Second, confidential data may be *encrypted* before it is sent on a public channel. Third, a *firewall* may protect the file server to restrict access from outside the subnetwork. While useful security building blocks, these standard techniques alone

* This research was supported by the Department of the Navy, Office of Naval Research, ONR Grant N00014-01-1-0968. Any opinions, findings, conclusions, or recommendations contained in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research. This work was supported, in part, by TFR, while the first author was at the Department of Computer Science, Chalmers University of Technology and University of Göteborg, Sweden. This work was supported, in part, by the German Research Foundation (DFG).

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 376–394, 2002.
© Springer-Verlag Berlin Heidelberg 2002

fail to guarantee that the confidentiality of data is preserved throughout the computation. The main reason is that these mechanisms ignore the *information flow* inside the program. Indeed, the file-client program that has legitimate access through the firewall, sends only encrypted data, and complies to access controls, may still leak information. For example, the program may contain the fragment if *fileScan*("foo") = *finApp* then *flag* := *true* where *flag* is a boolean variable (initialized to *false*) that signals whether to create a public temporary file containing a copy of the file "foo". If "foo" contains financial application data (*fileScan*("foo") = *finApp* holds), then this program successively surpasses the standard security mechanisms and introduces an undesired information flow. This is an example of an *implicit* [16] flow through a condition that depends on sensitive information (as opposed to an *explicit* flow performed by assigning sensitive data to a public variable). These leaks might be intended (when planted by a Trojan-horse program) or unintended (when caused by bugs, misconfiguration, debugging code, etc.). Syntactic scanning for malicious code (as realized by antivirus software) provides limited defense: rejecting a "black list" of syntactic patterns of known attacks does not preclude new attacks or semantic variations of the old ones.

An increasingly popular approach for identifying insecure information flow in programs is based on the notion of *noninterference* [20]. Suppose data, manipulated by a program, is partitioned into *high* (private) and *low* (public). According to noninterference, the program is secure iff high inputs do not interfere with low-observable behavior of the system (low outputs, timing, etc.). Originating from early work of Denning [15,17] and Cohen [13,14], a large body of work has followed the noninterference-based approach to confidentiality for various programming languages including [2,22,47,49,29,6,44,45,43,12,32,46].[1] We follow this line of work in our definition of security[2] for a language enriched with *message passing.*

Information flow in distributed languages. As computing systems become increasingly connected, *multi-threaded* and *distributed* programming languages become increasingly important [8]. Typical distributed languages heavily rely on message passing (e.g., Erlang [9] and Java [21]). Client-server applications are examples of message-passing-based programs. For instance, a distributed file-server program may create a new thread for every incoming request. Such a request is a message (passed by the client program) to open, read/write, and close a file. The server responds with messages that grant or deny these accesses. Due to the distribution, messages often travel over a publicly observable medium. This opens up opportunities for compromising the confidentiality of data. Not only may messages' contents reveal confidential data, but merely observing the presence of a message may lead to an undesirable leak. The latter possibility requires particular care when message contents are protected by *encryption.* Another source of leaks is *blocking,* which may change the observable behavior of a process that attempts to receive on an empty channel. In a similar fashion, the

[1] For motivating noninterference for confidentiality we refer to [35,50,45].

[2] Security is restricted to confidentiality in the rest if the article.

timing behavior of the program may cause a flow that the attacker can observe. Hence, we design a noninterference-based security specification that, in addition to explicit and implicit flows, rejects these other flows.

Static vs. dynamic confidentiality enforcement. Note that the above code can be modified as $flag := true$; if $fileType(\text{“foo”}) \neq finApp$ then $flag := false$ to perform the same leak. It is *non*execution of the assignment $flag := false$ that leads to this undesirable flow. As confidentiality, in general, is not a safety property [36, 48], the use of dynamic information-control mechanisms is rendered impractical as *all* potential execution paths must be monitored.

On the other hand, static analysis appears promising for enforcing security (e.g., [50,22,47,6,44,28,51,11,40]). Such an analysis is often formulated as a *security-type* system where security types correspond to the confidentiality level of data (such as high and low). A key contribution of this article is a security-type system that statically enforces confidentiality in a distributed system.

Overview. The rest of the paper is structured as follows. Section 2 explains main assumptions and features of the security model. Section 3 introduces a multi-threaded language with message passing. Section 4 presents a timing-sensitive security specification, scrutinizes the security of communication primitives wrt different types of channels, provides a series of examples, and concludes with compositionality results. Section 5 proposes a security-type system that enforces confidentiality. Section 6 gives an example of secure programming: a file server that relies on both multi-threading and message passing. Section 7 concludes.

2 Security Model

Distributed-language features have many implications for language security. The first step toward a tractable treatment is to clarify our assumptions behind the security model and motivate the model's intended features.

Compositionality. A prominent feature of a line of work [44,45,43] on noninterference for programs is the *compositionality* (or *hook-up* [34,36,31]) properties of security definitions. Such a property guarantees that when secure programs are plugged into an appropriate context then the resulting program must be secure. Hook-up properties provide an important foundation for modular system design. Thus, we aim to construct a compositional security property for a message-passing-enabled language.

Timing-sensitive security. Multi-threadedness (assuming a shared memory and execution on a single processor) has been a major focus of research in the context of noninterference-based confidentiality [22,47,49,44,43,12,32,46]. Common to these studies is the observation that if a program's *timing* behavior depends on high data, then the *scheduler* may reflect this dependence on the values of low variables. Suppose h and l are high and low variables, i.e., variables that initially store high and low data, respectively. Consider the program if $h =$

1 then C_{long} else skip where C_{long} is a time-consuming computation. Clearly, the program's timing behavior may reveal information about h. This is an example of a *timing flow* [27]. Moreover, in a multi-threaded setting this kind of leak may be encoded into a program that leaks h to l. Indeed, suppose $\|$ separates two threads in the two-threaded program (if $h = 1$ then C_{long} else skip); $l := 1 \quad \| \quad l := 0$. Under a typical fair scheduler (e.g., round-robin), if h is 1 then it is likely that the assignment $l := 0$ will have been executed before the assignment $l := 1$; and, thus, the value of h is likely to be copied to l. Note that the programs $l := 1$ and $l := 0$ are intuitively secure. Thus, accepting the original program as secure implies discarding the vital hook-up properties. This example illustrates that timing behavior, as observable by other threads, is an important ingredient for the security definition.

Our timing model follows previous work [44,32,43] in the assumption that the execution of each computation step takes a single unit of time. While this approach only roughly approximates the real timing behavior (which may depend on implementation- and hardware-specific factors as, e.g., caching [6]), it captures scheduler-based flows (as above) for a wide class of schedulers [44].

Distribution. In this article, we assume that threads are sequential programs; and multi-threading occurs at the level of local computation that operates on a shared memory. On the other hand, *processes* are potentially distributed such that each process has its own memory. Processes communicate by a communication network (among local computations) exchanging messages. Each process is potentially a multi-threaded program. Messages can be put onto a *channel* by sending commands and be taken out of a channel by receiving commands. Each channel is modeled by an *unordered list* (or a *finite multiset*) of messages, which adequately reflects the assumption that, in a distributed system, the order in which messages are delivered is not guaranteed.[3] For simplicity, we do not impose a particular discipline on using channels. However, depending on an application, it might be sensible to assume that, e.g., only one process can receive on a channel; or that a channel can be used only for point-to-point communication. Our security condition is independent of these assumptions.

The adversary. We assume that communication channels are partitioned into low, encrypted, and high channels. *Low* channels are observable by the attacker. Communication on low channels corresponds to, e.g., communication using standard Internet protocols such as TCP/IP and HTTP. Here, the traffic is vulnerable to eavesdropping. *Encrypted* channels are partly observable by the attacker. Namely, the attacker may observe the presence of a message, but not its contents. We adopt the (common) assumption that messages on such a channel are encrypted by an unbreakable encryption algorithm; and keys involved cannot be compromised. (Ongoing research addresses cryptographic aspects of formal models of encryption [5,28,39].) Finally, *high* channels are secure connections between processes that are invisible to the attacker. Communication on high

[3] An alternative assumption that a channel is a FIFO queue may be adequate for modeling communication channels at a high level of abstraction (e.g., pipelines in Unix). Our security specification can be easily adapted for this case (cf. [33]).

channels corresponds to, e.g., communication within a protected Intranet (an IP-based network of nodes behind a firewall). Here, the attacker cannot see the traffic.

The attacker has access to the low-observable part of the data and interprocess traffic. We assume that the machines in the network are trusted, although the code they run might come from an untrusted source. In this sense, the attacker is both *active*[4] (as a supplier of malicious code) and *passive* as an eavesdropper on the network. Our security-enforcement mechanism will reject potentially insecure programs from the active attacker so that running the code will not leak sensitive information to the passive attacker.

3 Multi-threaded While-Language with Message Passing

This section presents the syntax and semantics of a simple multi-threaded language that conforms to the assumptions of the previous section.

Local computation. Syntax and semantics for local computations are adopted from [44]. The syntax of a *command* (or a *thread*) is given by the grammar in Figure 1. Let boolean expressions B range over $BOOL$, arithmetic expressions Exp range over EXP, variables var range over VAR, values val range over VAL, and commands C, D range over CMD. Let $\vec{C}$ denote a vector of commands of the form $\langle C_1 \ldots C_n \rangle$. Vectors $\vec{C}, \vec{D}$ range over $\vec{CMD} = \cup_{n \in \mathbb{N}} CMD^n$, the set of multi-threaded programs, and cid ranges over CID, the set of channel identifiers.

$$
\begin{aligned}
CMD ::= {} & \mathsf{skip} \mid VAR := EXP \mid CMD_1; CMD_2 \mid \mathsf{if}\ BOOL\ \mathsf{then}\ CMD_1\ \mathsf{else}\ CMD_2 \\
& \mid \mathsf{while}\ BOOL\ \mathsf{do}\ CMD \mid \mathsf{fork}(CMD, \vec{CMD}) \mid \mathsf{send}(CID, EXP) \\
& \mid \mathsf{receive}(CID, VAR) \mid \mathsf{if\text{-}receive}(CID, VAR, CMD_1, CMD_2)
\end{aligned}
$$

Fig. 1. Command syntax

A *configuration* $\langle\!| C, mem, \sigma |\!\rangle$ (or $\langle\!| \vec{C}, mem, \sigma |\!\rangle$) is a triple, consisting of a command $C \in CMD$ (or a vector of commands $\vec{C} \in \vec{CMD}$), a local memory $mem : VAR \to VAL$ and the third element σ that accounts for interprocess communication (described below). The memory mem is a finite mapping from variables to values. The set of variables is partitioned into high and low security classes. For simplicity (but without loss of generality), we will often assume that there is only one variable for each security class, h and l, respectively. Further, we define *low-equality* on memories by: $mem_1 =_L mem_2$ iff the values of respective low variables for mem_1 and mem_2 are the same.

Each program executes under a shared memory on a single processor (or in a single process) such that at most one thread is active at any given point of

[4] We do not consider *integrity* properties and thus active attackers faking messages are not relevant.

$$[\text{Seq}_1] \quad \frac{\langle\!\langle C_1, mem, \sigma\rangle\!\rangle \rightarrowtriangle \langle\!\langle \langle\rangle, mem', \sigma\rangle\!\rangle}{\langle\!\langle C_1; C_2, mem, \sigma\rangle\!\rangle \rightarrowtriangle \langle\!\langle C_2, mem', \sigma\rangle\!\rangle}$$

$$[\text{Seq}_2] \quad \frac{\langle\!\langle C_1, mem, \sigma\rangle\!\rangle \rightarrowtriangle \langle\!\langle C_1'\vec{D}, mem', \sigma\rangle\!\rangle}{\langle\!\langle C_1; C_2, mem, \sigma\rangle\!\rangle \rightarrowtriangle \langle\!\langle (C_1'; C_2)\vec{D}, mem', \sigma\rangle\!\rangle}$$

$$[\text{Fork}] \quad \langle\!\langle \mathsf{fork}(C, \vec{D}), mem, \sigma\rangle\!\rangle \rightarrowtriangle \langle\!\langle C\vec{D}, mem, \sigma\rangle\!\rangle$$

$$[\text{Send}] \quad \frac{Exp \downarrow^{mem} val \qquad vals = \sigma(cid)}{\langle\!\langle \mathsf{send}(cid, Exp), mem, \sigma\rangle\!\rangle \rightarrowtriangle \langle\!\langle \langle\rangle, mem, \sigma[cid \mapsto val.vals]\rangle\!\rangle}$$

$$[\text{Rcv}] \quad \frac{\sigma(cid) = vals.val.vals'}{\langle\!\langle \mathsf{receive}(cid, var), mem, \sigma\rangle\!\rangle \rightarrowtriangle \langle\!\langle \langle\rangle, mem[var \mapsto val], \sigma[cid \mapsto vals.vals']\rangle\!\rangle}$$

$$[\text{IfRF}] \quad \frac{\sigma(cid) = \langle\rangle}{\langle\!\langle \mathsf{if\text{-}receive}(cid, var, C_1, C_2), mem, \sigma\rangle\!\rangle \rightarrowtriangle \langle\!\langle C_2, mem, \sigma\rangle\!\rangle}$$

$$[\text{IfRT}] \quad \frac{\sigma(cid) = vals.val.vals'}{\langle\!\langle \mathsf{if\text{-}receive}(cid, var, C_1, C_2), mem, \sigma\rangle\!\rangle \rightarrowtriangle \langle\!\langle C_1, mem[var \mapsto val], \sigma[cid \mapsto vals.vals']\rangle\!\rangle}$$

Fig. 2. Small-step deterministic semantics of commands

$$[\text{Pick}] \quad \frac{\langle\!\langle C_i, mem, \sigma\rangle\!\rangle \rightarrowtriangle \langle\!\langle \vec{C}, mem', \sigma'\rangle\!\rangle}{\langle\!\langle \langle C_1 \ldots C_n\rangle, mem, \sigma\rangle\!\rangle \rightarrow \langle\!\langle \langle C_1 \ldots C_{i-1}\vec{C}C_{i+1} \ldots C_n\rangle, mem', \sigma'\rangle\!\rangle}$$

$$[\text{Step}] \quad \frac{\langle\!\langle \vec{C}_k, mem_k, \sigma\rangle\!\rangle \rightarrow \langle\!\langle \vec{C}_k', mem_k', \sigma'\rangle\!\rangle}{\lhd(\vec{C}_1, mem_1), \ldots, (\vec{C}_n, mem_n); \sigma\rhd \twoheadrightarrow \lhd(\vec{C}_1, mem_1), \ldots, (\vec{C}_k', mem_k'), \ldots, (\vec{C}_n, mem_n); \sigma'\rhd}$$

Fig. 3. Concurrent semantics of programs

$$\begin{aligned}
\mathsf{synch\text{-}send}(cid, Exp) &= \mathsf{receive}(destReady[cid], readyVar); \mathsf{send}(cid, Exp); \\
&\quad\ \mathsf{receive}(destAck[cid], ackVar) \\
\mathsf{synch\text{-}receive}(cid, var) &= \mathsf{send}(destReady[cid], \text{“ready”}); \mathsf{receive}(cid, var); \\
&\quad\ \mathsf{send}(destAck[cid], \text{“got it”})
\end{aligned}$$

Fig. 4. Synchronous communication primitives

time. The small-step semantics is given by transitions between configurations. The deterministic part of the semantics is defined by the transition rules in Figure 2, omitting natural rules for skip, assignment, if and while for lack of space (cf. [44]). Arithmetic and boolean expressions are executed atomically by $\downarrow$ transitions. The local $\rightarrowtriangle$-transitions are deterministic. The general form of

a deterministic transition is either $\langle\!\langle C, mem, \sigma \rangle\!\rangle \rightarrowtriangle \langle\!\langle \langle\rangle, mem', \sigma' \rangle\!\rangle$, which means termination with the final memory mem', or $\langle\!\langle C, mem, \sigma \rangle\!\rangle \rightarrowtriangle \langle\!\langle C'\vec{D}, mem', \sigma' \rangle\!\rangle$. Here, one step of computation starting with command C in a memory mem gives a new main thread C', a vector $\vec{D}$ of spawned threads, a new memory mem' and new channel status σ'. The command $\mathsf{fork}(C, \vec{D})$ dynamically creates a new vector $\vec{D}$ of threads that run in parallel with the main thread C. This has the effect of adding the vector $\vec{D}$ to the configuration.

The rule [Pick] in Figure 3 defines the concurrent semantics within a program. Whenever the scheduler picks a thread C_i for execution, a $\rightarrow$-transition takes place updating the command pool and the rest of the configuration according to a (small) computation step of C_i.

Communication primitives and global computation. We now present the communication primitives and global semantics, which we adapt from [33]. Let channel contents range over *CHVAL*, the set of unordered lists over *VAL*. Given a channel id *cid*, the *channel status function* $\sigma : CID \rightarrow CHVAL$ returns the unordered list of messages that are currently waiting on *cid*.

The command $\mathsf{send}(cid, Exp)$ sends the value of an expression *Exp* on a channel *cid*. We distinguish two receiving primitives. *Blocking* $\mathsf{receive}(cid, var)$ blocks until it receives a value on the channel *cid*. Once the value is received, the variable *var* is set to that value. *Nonblocking* receive $\mathsf{if\text{-}receive}(cid, var, C_1, C_2)$ always continues execution. If the channel *cid* is nonempty then *var* is set to the received value and execution continues with the command C_1. Otherwise execution continues with the command C_2. Note that the order of delivery does not necessarily coincide with the order in which messages were sent. This accurately models our assumption (cf. Section 2) implied by the distributed nature of the system. Note that the primitive send is *asynchronous* in the sense that it does not wait for the message to be received. A *synchronous* version synch-send blocks until the message has been received by a synch-receive on the same channel. Synchronous communication (synch-send and synch-receive) can be achieved by using asynchronous communication primitives. Figure 4 displays how this can be done in the simple case of at most one sender and at most one receiver for each channel $cid \in CID$ (cf. [8]). It uses auxiliary arrays of channels *destReady* and *destAck*. Care must be taken when constructing such an implementation when multiple readers/writers are possible. (For example, the encoding of Figure 4 is then not suitable as a *destAck* message may be received by a sender whose message is still in transit). For our purposes, the key observation is that both synch-send and synch-receive use receive in their implementations.

Given a (possibly distributed) collection of programs $\vec{C_1}, \ldots, \vec{C_n}$, in which each program executes on its own memory $mem_1, \ldots, mem_n$, respectively, a *global* configuration $\lhd(\vec{C_1}, mem_1), \ldots, (\vec{C_n}, mem_n); \sigma \rhd$ consists of two components. The first component is a sequence of pairs each containing a program and its memory. The second component is the channel status function σ. Nondeterministic $\twoheadrightarrow$-transitions on global configurations are defined by the rule [Step] in Figure 3. The rule [Step] is similar to the rule [Pick]. It ensures that a global transition takes place whenever a local transition occurs.

4 Security Specification

Recall that noninterference means that low-observable behavior is unchanged as high inputs are varied. The indistinguishability of behavior for the attacker can be represented naturally by the notion of *bisimulation* (e.g, [18,44,46,12]). Following our justification of timing-sensitive security in the context of concurrent and distributed programming in Section 2, we adopt the timing-sensitive *strong low-bisimulation* [44] that can be defined on multi-threaded programs with [43] and without synchronization [44]. If two commands might have a different timing behavior depending on high data then they are not low-bisimilar. This definition has been extended for a language with message passing, but only with high and low channels for communication [33].

4.1 Timing-Sensitive Security Definition

In order to lift low-bisimulation to handle encrypted channels we extend low-equality $=_L$ (defined earlier on memories) to relate channel status functions that agree on their low-observable arguments. Let $dom : CID \to \{high, enc, low\}$ be a function that given a channel id cid returns its security level. Formally, define for $\sigma_1, \sigma_2 : CID \to CHVAL$:

$$\sigma_1 =_L \sigma_2 \iff (\forall cid \in CID.\ dom(cid) = low \Longrightarrow \sigma_1(cid) = \sigma_2(cid)\ \wedge \\ dom(cid) = enc \Longrightarrow |\sigma_1(cid)| = |\sigma_2(cid)|)$$

where $|\sigma(cid)|$ is the number of messages on channel cid. This suits our assumption that for the attacker: (i) low channels are fully observable, (ii) only the size of encrypted channels is observable, and (iii) *high* channels are not observable.

Definition 1 *Define* strong low-bisimulation $\approxeq_L$ *to be the union of all symmetric relations R on command pools (programs) of equal size for which whenever $\langle C_1 \ldots C_n \rangle\ R\ \langle D_1 \ldots D_n \rangle$ then for all $mem_1, mem_2, \sigma_1, \sigma_2, i$ we have*

$$\langle\!\langle C_i, mem_1, \sigma_1 \rangle\!\rangle \rightarrowtriangle \langle\!\langle \vec{C'}, mem'_1, \sigma'_1 \rangle\!\rangle \wedge mem_1 =_L mem_2 \wedge \sigma_1 =_L \sigma_2 \Longrightarrow \exists \vec{D'}, mem'_2, \sigma'_2.$$
$$\langle\!\langle D_i, mem_2, \sigma_2 \rangle\!\rangle \rightarrowtriangle \langle\!\langle \vec{D'}, mem'_2, \sigma'_2 \rangle\!\rangle \wedge mem'_1 =_L mem'_2 \wedge \sigma'_1 =_L \sigma'_2 \wedge \vec{C'}\ R\ \vec{D'}$$

Intuitively, two programs $\langle C_1 \ldots C_n \rangle$ and $\langle D_1 \ldots D_n \rangle$ are low-bisimilar iff (i) they are command vectors of the same size and (ii) for each pair of respective commands C_i and D_i occurring in the same position i, varying the high parts of memories, the contents of encrypted channels (but not size) and the contents of high channels (including the size) at any point of a computation does not introduce any difference between the low parts of the memories, the sizes of encrypted channels and the contents of low channels throughout the computation. Despite any high variation such two commands will still execute in lock step, i.e., their timing behavior is the same. We are ready to state the security specification.

Definition 2 *A program $\vec{C}$ is* secure *if and only if $\vec{C} \approxeq_L \vec{C}$.*

While one can prove that the relation $\approxeq_L$ is transitive, it is not reflexive. For example, the insecure program $l := h$ is not $\approxeq_L$-related to itself, as the low-equality of memories can be broken by a computation step.

4.2 Security of Communication Primitives

This section is devoted to the communication primitives and their effect on security. Of particular interest is the identification of *secure contexts* for hook-up properties (cf. Section 2) involving communication primitives. Inserting secure programs in such a context should result in a secure program. In the next section, we will derive a security-type system from the hook-up properties. Define an expression Exp to be *low* iff $\forall mem_1, mem_2.\, mem_1 =_L mem_2 \Longrightarrow \exists n.\, Exp \downarrow^{mem_1} n \wedge Exp \downarrow^{mem_2} n$. Otherwise, the expression is *high*. In the rest of this section, Exp_h ranges over high expressions while Exp ranges over arbitrary expressions.

Send. Sending on either a high or an encrypted channel is a secure command, because send is nonblocking. Indeed, executing $\mathsf{send}(cid, Exp)$ after varying the high part of the memory and channel status function will always result in low-equal memories and low-equal channel status functions. This is true for both high and encrypted channels: low-level observations remain unchanged in the case of sending on a high channel. In the case of an encrypted channel, the size of the channel will always be increased by 1 independently of the variation of high data. To show that such a program is secure we, according to Definition 1, need to construct a symmetric relation that makes this program low-bisimilar to itself. One suitable relation is the relation $\{(\mathsf{send}(cid, Exp), \mathsf{send}(cid, Exp)), (\langle\rangle, \langle\rangle)\}$. This proves that $\mathsf{send}(cid, Exp)$ is a secure ground context for any expression Exp. This and following examples of communication primitives and their security are collected in Figure 7 in Appendix A.

However, if a send occurs in a branch of a *high conditional* (i.e., a conditional whose guard is a high expression), then sending on an encrypted channel in one branch and not sending in the other (e.g., $\mathsf{if}\ Exp_h\ \mathsf{then}\ \mathsf{send}(cid, Exp)\ \mathsf{else}\ \mathsf{skip}$) makes low-observable behavior different. Namely, the size of the channel will be different when varying the initial value of h. Formally, take mem_1 and mem_2 such that $mem_1 =_L mem_2$, $Exp_h \downarrow^{mem_1} false$, $Exp_h \downarrow^{mem_2} true$ (which is possible because Exp_h is high), and $\sigma = \lambda(cid).\langle\rangle$. Since $\langle\!\langle \mathsf{if}\ Exp_h\ \mathsf{then}\ \mathsf{send}(cid, Exp)\ \mathsf{else}\ \mathsf{skip}, mem_1, \sigma\rangle\!\rangle \rightarrow \langle\!\langle \mathsf{skip}, mem_1, \sigma\rangle\!\rangle$ and $\langle\!\langle \mathsf{if}\ Exp_h\ \mathsf{then}\ \mathsf{send}(cid, Exp)\ \mathsf{else}\ \mathsf{skip}, mem_2, \sigma\rangle\!\rangle \rightarrow \langle\!\langle \mathsf{send}(cid, Exp), mem_2, \sigma\rangle\!\rangle$ holds, for the initial program to be secure it must be the case that $\mathsf{skip} \approxeq_L \mathsf{send}(cid, Exp)$. However, we have $\langle\!\langle \mathsf{skip}, mem_1, \sigma\rangle\!\rangle \rightarrow \langle\!\langle \langle\rangle, mem_1, \sigma\rangle\!\rangle$ and $\langle\!\langle \mathsf{send}(cid, Exp), mem_2, \sigma\rangle\!\rangle \rightarrow \langle\!\langle \langle\rangle, mem_2, \sigma[cid \mapsto n\langle\rangle]\rangle\!\rangle$, where $Exp \downarrow^{mem_2} n$. Clearly, $\sigma \neq_L \sigma[cid \mapsto n\langle\rangle]$ for any n because $dom(cid) = enc$. Note that the same program for a high channel is perfectly secure.

A similar argument explains why sending on different channels depending on a high condition (e.g., $\mathsf{if}\ Exp_h\ \mathsf{then}\ \mathsf{send}(cid, Exp)\ \mathsf{else}\ \mathsf{send}(cid', Exp')$) is insecure. Only when sending (possibly different) values on the same channel in both branches, may a command be secure (e.g., $\mathsf{if}\ Exp_h\ \mathsf{then}\ \mathsf{send}(cid,)$ $Exp\ \mathsf{else}\ \mathsf{send}(cid, Exp')$) for an encrypted channel. Sending Exp_h on a low channel cl is, naturally, insecure (e.g., $\mathsf{send}(cl, Exp_h)$).

Receive. Receiving on a high channel (e.g., $\mathsf{receive}(cid, h)$) is insecure as it may introduce blocking if the channel is empty. On the other hand, receiving on an encrypted channel is secure because the number of messages on an encrypted

channel is the same for two low-equal channel status functions. As is the case with conditional sending, conditional receiving is insecure for encrypted channels (e.g., if Exp_h then receive(cid, h) else skip and also if Exp_h then receive(cid, h) else receive(cid', h)) unless the same channel is used (e.g., if Exp_h then receive(cid, h) else receive(cid, h')). Using a low variable for storing the data received on a high or encrypted channel (e.g., receive(cid, l)) is, of course, insecure.

Nonblocking receive if-receive(cid, h, C_1, C_2) on an encrypted channel is secure provided both branches C_1 and C_2 are secure programs. Indeed, what branch was taken is observable by the attacker because it is observable whether the channel cid is empty. In case cid is high, the stronger condition $C_1 \approxeq_L C_2$ (which implies that C_1 and C_2 are secure by the transitivity of the relation $\approxeq_L$) is imposed to ensure the security of the overall program. Such a condition guarantees that the behavior of the branches is indistinguishable for the attacker.

Synchronous communication. Both synch-send and synch-receive inherit their security properties from the receive that is used in their implementations. This means that neither synchronous sending nor receiving is secure on a high channel. For an encrypted channel cid, the programs synch-send(cid, h), synch-receive(cid, h), if Exp_h then synch-send(cid, Exp) else synch-send(cid, Exp'), and if Exp_h then synch-receive(cid, h) else synch-receive(cid, h') are secure.

4.3 Lessons Learned

Essential implications for the programmer from our consideration of the security of communication primitives wrt different types of channels can be briefly summarized as follows:

- High channels allow for liberal use of send in the branches of high conditionals. However, receive on a high channel is not secure due to the potential introduction of blocking. As a result, the programmer is restricted to the nonblocking if-receive which masks the presence of messages on the channel.
- Encrypted channels are appropriate for both send and receive, but restrictive on high conditionals. Thus, when the occurrence of a communication of some high data does not depend on any high data, it is an appropriate channel for an efficient implementation. Indeed, as receive is a secure primitive, there is no need for the busy-waiting loops with the if-receive primitive.
- if-receive is a secure primitive for both high and encrypted channels, but for encrypted channels receive is preferable over the inefficient busy-waiting loops with if-receive.
- Each of the synchronous primitives synch-send and synch-receive inherit their restrictiveness from receive. This implies that neither synchronous sending nor synchronous receiving is a secure command for a high channel.
- Confidentiality might be compromised if communication primitives operating on encrypted or low channels are used in the branches of a high if or if-receive that receives on a high channel. This is similar to implicit flows that are the effect of an assignment to l in the branches of a high if.

4.4 Hook-Up Properties

Supported by the intuition of Sections 4.2 and 4.3, we present the compositionality result for the complete language. This result extends the secure congruence theorem for a multi-threaded language [44] to contexts that preserve security in the presence of message passing. The proof of the hook-up result is conducted by the "up to" technique [37]. Let $[\bullet]$ be a hole for a command. A context $\mathbb{C}[\bullet_1, \ldots, \bullet_n]$ for some $n \geq 2$ is *secure* iff it has one of these forms:

$$
\begin{aligned}
\mathbb{C}[\bullet_1, \ldots, \bullet_n] ::= {} & \mathsf{skip} \mid h := Exp \mid l := Exp \ (Exp \text{ is low}) \mid [\bullet_1]; [\bullet_2] \\
& \mid \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]\ (B \text{ is low}) \mid \mathsf{while}\ B\ \mathsf{do}\ [\bullet_1]\ (B \text{ is low}) \\
& \mid \mathsf{fork}([\bullet_1], \langle [\bullet_2] \ldots [\bullet_n] \rangle) \mid \mathsf{send}(cid, Exp)\ (dom(cid) = high) \\
& \mid \mathsf{send}(cid, Exp),\ \mathsf{receive}(cid, h)\ (dom(cid) = enc) \\
& \mid \mathsf{send}(cid, Exp),\ \mathsf{receive}(cid, var)\ (dom(cid) = low,\ Exp \text{ is low}) \\
& \mid \mathsf{if\text{-}receive}(cid, h, [\bullet_1], [\bullet_2])\ (dom(cid) = enc) \\
& \mid \mathsf{if\text{-}receive}(cid, var, [\bullet_1], [\bullet_2])\ (dom(cid) = low)
\end{aligned}
$$

Theorem 1 (Hook-up) *If for some $n \geq 2$ the context $\mathbb{C}[\bullet_1, \ldots, \bullet_n]$ is a secure context and $C_1, \ldots, C_n$ are secure then $\mathbb{C}[C_1, \ldots, C_n]$ is secure. If $\mathbb{C}[\bullet_1, \bullet_2] =$ if B then $[\bullet_1]$ else $[\bullet_2]$ (B is high) or $\mathbb{C}[\bullet_1, \bullet_2] =$ if-receive$(cid, h, [\bullet_1], [\bullet_2])$ (such that $dom(cid) = high$), then $\mathbb{C}[C_1, C_2]$ is secure provided $C_1 \approxeq_L C_2$.*

5 Type-Based Security Analysis

This section presents an automatic compositional analysis for enforcing program confidentiality, extending previous approaches [6,44] to handle communication primitives. Due to the compositional nature of the type system, both designing it and proving it to be sound are greatly facilitated by the hook-up results.

The type system. The analysis is based on a type system that transforms a given program into a new program. Either the initial program is rejected (due to explicit, implicit or other insecure information leaks) or it might be accepted by the system and transformed into a program that is also free of timing leaks. The transformation rules have the form $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$, where $\vec{C}$ is a program, $\vec{C'}$ is the result of its transformation and $\vec{Sl}$ is the type of $\vec{C'}$. The type $\vec{Sl}$ is $\vec{C'}$'s *low slice*. The low slice $\vec{Sl}$ has no occurrences of h and models the timing behavior of $\vec{C'}$, as observable by other threads/processes.

Any expression can by typed *high* (as it might depend on h). On the other hand, only expressions that have no occurrences of h are typed *low*. (These expressions can be safely used in assignments to l.) Typing rules for expressions and communication primitives are presented in Figure 5. Typing rules for other language constructs are like in [44]. The rule $[\mathrm{S}_{high}]$ types a send on a high channel with the low slice skip. Although receive on a high channel is not typable, if-receive

$$[\text{Exp}] \quad Exp : high \qquad \frac{h \notin Vars(Exp)}{Exp : low}$$

$$\text{---------- } \textbf{\textit{cid}} \textbf{ is } \textbf{\textit{high}}$$

$$[\text{S}_{high}] \quad \mathsf{send}(cid, Exp) \hookrightarrow \mathsf{send}(cid, Exp) : \mathsf{skip}$$

$$[\text{IR}_{high}] \quad \frac{C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2 \quad al(Sl_1) = al(Sl_2) = false}{\mathsf{if\text{-}receive}(cid,h,C_1,C_2) \hookrightarrow \mathsf{if\text{-}receive}(cid,h,C_1';Sl_2,Sl_1;C_2') : \mathsf{skip}; Sl_1; Sl_2}$$

$$\text{---------- } \textbf{\textit{cid}} \textbf{ is } \textbf{\textit{enc}}$$

$$[\text{S}_{enc}] \quad \mathsf{send}(cid, Exp) \hookrightarrow \mathsf{send}(cid, Exp) : \mathsf{send}(cid, 0)$$

$$[\text{R}_{enc}] \quad \mathsf{receive}(cid, h) \hookrightarrow \mathsf{receive}(cid, h) : \mathsf{receive}(cid, \hat{h})$$

$$[\text{IR}_{enc}] \quad \frac{C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{\mathsf{if\text{-}receive}(cid,h,C_1,C_2) \hookrightarrow \mathsf{if\text{-}receive}(cid,h,C_1',C_2') : \mathsf{if\text{-}receive}(cid,\hat{h},Sl_1,Sl_2)}$$

$$[\text{SS}_{enc}] \quad \mathsf{synch\text{-}send}(cid, Exp) \hookrightarrow \mathsf{synch\text{-}send}(cid, Exp) : \mathsf{synch\text{-}send}(cid, 0)$$

$$[\text{SR}_{enc}] \quad \mathsf{synch\text{-}receive}(cid, h) \hookrightarrow \mathsf{synch\text{-}receive}(cid, h) : \mathsf{synch\text{-}receive}(cid, \hat{h})$$

$$\text{---------- } \textbf{\textit{cid}} \textbf{ is } \textbf{\textit{low}}$$

$$[\text{S}_{low}] \quad \frac{Exp : low}{\mathsf{send}(cid, Exp) \hookrightarrow \mathsf{send}(cid, Exp) : \mathsf{send}(cid, Exp)}$$

$$[\text{R}_{low}] \quad \mathsf{receive}(cid, var) \hookrightarrow \mathsf{receive}(cid, var) : \mathsf{receive}(cid, \hat{var})$$

$$[\text{IR}_{low}] \quad \frac{C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{\mathsf{if\text{-}receive}(cid,var,C_1,C_2) \hookrightarrow \mathsf{if\text{-}receive}(cid,var,C_1',C_2') : \mathsf{if\text{-}receive}(cid,\hat{var},Sl_1,Sl_2)}$$

$$[\text{SS}_{low}] \quad \frac{Exp : low}{\mathsf{synch\text{-}send}(cid, Exp) \hookrightarrow \mathsf{synch\text{-}send}(cid, Exp) : \mathsf{synch\text{-}send}(cid, Exp)}$$

$$[\text{SR}_{low}] \quad \mathsf{synch\text{-}receive}(cid, var) \hookrightarrow \mathsf{synch\text{-}receive}(cid, var) : \mathsf{synch\text{-}receive}(cid, \hat{var})$$

Fig. 5. Security typing of expressions and communication primitives

can be typed after cross-copying the branches (as for if on a high conditional [6,44]) to equalize the timing behavior independently whether the channel *cid* is empty (the rule [IR_{high}]). To (conservatively) prevent potential implicit flows, no assignment to l or communication on encrypted or low channels is allowed in the branches. Let $al(C)$ be a boolean function returning *true* whenever there is a syntactic occurrence of either an assignment to l or a communication primitive on encrypted or low channels. The condition $al(Sl_1) = al(Sl_2) = false$ prevents implicit leaks. The slice of the overall command is the sequential composition of

the slices of the branches prefixed with a skip corresponding to the time tick for the guard inspection.

Typing communication primitives on encrypted channels is compositional, i.e., the type (slice) of a program is assembled from the types (slices) of the program's components. Although the low slices of the typing cannot use h, they still must expose the same behavior wrt $|cid|$. This is achieved by using 0 instead of the potentially high *Exp* in the slice in $[\mathrm{S}_{enc}]$ and $[\mathrm{SS}_{enc}]$, and using versions of receive, if-receive, synch-receive that do not update h in $[\mathrm{R}_{enc}]$, $[\mathrm{IR}_{enc}]$ and $[\mathrm{SR}_{enc}]$. For this purpose we use the notation $\hat{var}$ where $\hat{l} = l$ and $\hat{h} = \_$ to indicate the case that h is not updated after a reception. Note that the generation of such slices does not change the semantics of the original program. Indeed, no low slice Sl created in $[\mathrm{S}_{enc}]$ or $[\mathrm{SS}_{enc}]$ is used in cross-copying rules because $al(Sl) = true$. Directed by the hook-up result, we proceed analogously to derive the types of communication primitives on low channels.

Correctness of the analysis. The correctness of the analysis follows from the observation that the attacker cannot distinguish the result of the transformation and its slice. In other words, $\vec{C} \hookrightarrow \vec{C}' : \vec{Sl}$ implies $\vec{C}' \approxeq_L \vec{Sl}$, which can be proved by the hook-up theorem. By the transitivity of $\approxeq_L$ we have $\vec{C}' \approxeq_L \vec{C}'$. This leads us to the correctness theorem.

Theorem 2 (Correctness of the Analysis) $\vec{C} \hookrightarrow \vec{C}' : \vec{Sl} \Longrightarrow \vec{C}'$ *is secure.*

6 Example of Secure Programming: A File Server

The correctness theorem guarantees robust timing-sensitive security, but are there any useful secure programs? While we refer to [7] for efficient secure algorithms (such as sorting and searching) in a sequential setting, let us return to the distributed and multi-threaded file server and client programs and sketch how they can be typed.

Consider the program fragments in Figure 6. The multi-threaded file server *ServerMan* dynamically creates threads $Server[i]$ ($i \in \mathbb{N}$) upon new requests from a file client $Client[j]$ with the thread identifier $j \in \mathbb{N}$. The purpose of the file server is to store confidential data and to support fast accesses over a network that might be audited by attackers. Hence, the use of encrypted channels is a natural necessity.[5] The channels *open*, $open[i]$, $openReply[i]$, $access[i]$ are encrypted channels. Thus, assuming the variables *fileName*, *fname*, *accessArgs*, *request*, *results* are *high*, these programs can be typable and, thus, are secure.[6]

This example illustrates that our security condition and type system, although being rigorous enough to ensure security, are not over-restrictive. Namely, useful programs that fulfill our conditions can be written. Note that using encrypted channels we are able to avoid a busy-loop waiting overhead (that would

[5] This is practiced, e.g., at the "privacy level" in the DCE/DFS distributed file system.

[6] Note that the example assumes an extension of our language with pairs and arrays. Security definition and the type system (along with the correctness proof) can be adjusted for these constructs following [6].

```
ServerMan = while true
              do receive(open, (fileName, clientID));
                 ... /* generate a fresh id i for the thread to be created */
                 fork(skip, Server[i]);   /* fork off a thread handling accesses */
                 send(open[i], (fileName, clientID))
Server[i] = receive(open[i], (fname, clid));
            ... /* open file fname; if successful then: */
            send(openReply[clid], i); more := true;
            while more
              do receive(access[i], request);
                 if action(request) = READ then ...
                 if action(request) = WRITE then ...
                 if action(request) = CLOSE then ... more := false
                 send(accessReply[clid], ...)   /* send access results */
Client[j] = send(open, ("foo", j)); receive(openReply[j], serverID);
            send(access[serverID], accessArgs); receive(accessReply[j], results)
```

Fig. 6. A multi-threaded file server and a file client

have been necessary, e.g., in [33]) and yet guarantee timing-sensitive security. Due to space limitations, the example focuses on only two primitives, send and receive. However, similar examples exist for the usability of if-receive, synch-send and synch-receive wrt our security condition.

7 Conclusions

It might seem astonishing that, in some cases, secure programming with protected (high) channels is more restrictive than with encrypted or cleartext (low) channels. However, this is the case for the receive primitive which is insecure for high channels but secure for encrypted and low channels. This phenomenon stems from the nonmonotonic nature of confidentiality properties: the fact that the attacker observes less about data does not imply that the data requires less (or, conversely, more) protection. Indeed, as can be seen from our security specification, as the attacker's observational power decreases, the range of possible values of sensitive data increases.

Contributions. In order to investigate the interplay between the observational power of the attacker and the expressive power of a programming language featuring multi-threading, (non)blocking and (a)synchronous message passing, we have proposed a timing-sensitive confidentiality specification. This specification

accommodates three types of communication channels (for high, encrypted and low data) and enjoys a hook-up property wrt a number of contexts. To the best of our knowledge, this article is the first to prove hook-up properties for timing-sensitive confidentiality in a distributed language. As an important contribution, we have derived a sound security-type system from the hook-up result. Essential implications for programming secure systems are summarized in Section 4.3.

This article follows [33] as a starting point, where security in a distributed language is defined. The novelties of this article wrt [33] include an investigation of encrypted channels, synchronous communication primitives, an analysis of what primitives can be used securely for what channels, a development of hook-up properties at the level of commands[7] and, most importantly, a security enforcement mechanism in the form of a security-type system.

Related work. As far as we are aware, Reitman's security logic [41] is the first to address message-passing primitives in the context of confidentiality. His primitives correspond to our asynchronous send and receive with only one generic type of a communication channel. Banâtre and Bryce's security logic [10] describes security properties for a language based on CSP [24]. Their primitives correspond to our synchronous send and receive and, thus, their use is severely restricted. Both of these logics lack noninterference proofs and automatic inference algorithms. Furthermore, they do not treat timing flows. Mizuno and Oldehoeft [38] consider information flow in an object-oriented distributed system. The lack of noninterference results and a costly runtime security mechanism (that is invoked every time a message is sent) are major drawbacks of this work.

Abadi and Blanchet [1,3] have devised type systems that guarantee confidentiality for a calculus of cryptographic protocols, the spi calculus [4]. This approach is capable of handling key-exchange protocols. However, their formalism is specific to the spi-calculus: secret keys and their usage are hidden from the low-observable view so that the resulting security condition allows for protocols with encryption to be represented in a noninterference-like fashion.

Honda et al. propose a powerful type system for tracking information flow in the asynchronous pi-calculus [25,26]. While not designed for expressing information flow in high-level languages, their rich *channel types* allow for encoding other type systems (e.g., [47]) for high-level multi-threaded languages.

Zdancewic et al. introduce *secure program partitioning* as a technique for secure distributed programming on heterogeneously trusted hosts [52]. This approach is appealing because it addresses the interaction between confidentiality and integrity in a distributed system. However, no convincing security assurance results are available for secure program partitioning.

There is a large body of work on noninterference in the setting of process algebras (see [18,42] for an overview) and other event-based systems (see, e.g., [30] for an overview). Common ground with many process algebra-based security specifications and our study is bisimulation-based security and an asynchronous

[7] The compositionality principle of [33] focuses on the level of processes rather than on the level of commands. It asserts that the composition of secure processes results in an overall system that satisfies a global trace-based security property.

semantics of computation (which allows for modeling subcomputations with different relative speeds). However, our primary focus is different in considering (i) a concrete programming language, (ii) a timing-sensitive security specification and (iii) asynchronous message passing. Although some recent process-algebra-based investigations have explored timing-sensitive security (e.g., [19]) and security in an asynchronous setting (e.g., [23,26]), timing is often disregarded; and handshaking models that require both sides to synchronize during communication are typically used. There is no such requirement under asynchronous message passing (as expressed by send, receive and if-receive), which is suitable for modeling programming in a distributed environment.

Future work. While we have assumed that programs are executed on trusted hosts, some machines may be entirely controlled by the attacker in a realistic distributed setting. Thus, an important goal for future work is to obtain a confidentiality specification that is sensitive to malicious hosts.

It is essential to lift another practically unrealistic assumption—that all messages are reliably delivered. We expect that both the security condition and the type system are robust wrt message loss. Indeed, assuming communication failures do not depend on sensitive data, we can model a lossy channel *cid* by the process $fail(cid) = \mathsf{while}\ true\ \mathsf{do}\ \mathsf{if\text{-}receive}(cid, x_{cid}, \mathsf{skip}, \mathsf{skip})$ for a fresh high variable x_{cid}. That the security of a system cannot be compromised by such a failure process is a simple corollary of the hook-up result and the observation that $fail(cid)$ is secure for any *cid*.

Acknowledgments Thanks to Dan Grossman, Andrew Myers, David Sands, Fred Schneider, Peter Sewell, and Steve Zdancewic for insightful feedback.

References

1. M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999.
2. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. of Symposium on Principles of Programming Languages*, January 1999.
3. M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In *FOSSACS'01*, *LNCS 2030*, pages 25–41, April 2001.
4. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proc. of Conference on Computer and Communications Security*, 1997.
5. M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2), 2002.
6. J. Agat. Transforming out timing leaks. In *Proc. of Symposium on Principles of Programming Languages*, pages 40–53, January 2000.
7. J. Agat and D. Sands. On confidentiality and algorithms. In *Proc. of IEEE Symposium on Security and Privacy*, May 2001.
8. G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley, 2000.
9. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang.* Prentice-Hall, 2nd edition, 1996.

10. J.-P. Banâtre and C. Bryce. Information flow control in a parallel language framework. In *Proc. of IEEE Computer Security Foundations Workshop*, June 1993.
11. A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proc. of IEEE Computer Security Foundations Workshop*, June 2002.
12. G. Boudol and I. Castellani. Noninterference for concurrent programs. In *ICALP'01*, *LNCS 2076*, pages 382–395, July 2001.
13. E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.
14. E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
15. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
16. D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
17. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.
18. R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995.
19. R. Focardi, R. Gorrieri, and F. Martinelli. Information flow analysis in a discrete-time process algebra. In *Proc. of IEEE Computer Security Foundations Workshop*, pages 170–184, July 3–5 2000.
20. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. of IEEE Symposium on Security and Privacy*. April 1982.
21. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
22. N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. of Symposium on Principles of Programming Languages*, 1998.
23. M. Hennessy and J. Riely. Information flow vs resource access in the asynchronous pi-calculus (extended abstract). In *ICALP'00*, *LNCS 1853*, pages 415–427, 2000.
24. C. Hoare. *Communicating Sequential Processes (CSP)*. Prentice Hall, 1985.
25. K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proc. of European Symposium on Programming*, *LNCS 1782*, pages 180–199, 2000.
26. K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proc. of Symposium on Principles of Programming Languages*, January 2002.
27. B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, Oct. 1973.
28. P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. of European Symposium on Programming*, *LNCS 2028*, April 2001.
29. K. R. M. Leino and R. Joshi. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.
30. H. Mantel. Possibilistic definitions of security – an assembly kit –. In *Proc. of IEEE Computer Security Foundations Workshop*, pages 185–199, July 2000.
31. H. Mantel. On the composition of secure systems. In *Proc. of IEEE Symposium on Security and Privacy*, May 2002.
32. H. Mantel and A. Sabelfeld. A generic approach to the security of multi-threaded programs. In *Proc. of IEEE Computer Security Foundations Workshop*, June 2001.
33. H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security*, 2002. To appear.

34. D. McCullough. Specifications for multi-level security and hook-up property. In *Proc. of IEEE Symposium on Security and Privacy*, pages 161–166. May 1987.
35. J. McLean. The specification and modeling of computer security. *Computer*, 23(1), January 1990.
36. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. of IEEE Symposium on Security and Privacy*, 1994.
37. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
38. M. Mizuno and A. Oldehoeft. Information flow control in a distributed object-oriented system with statically-bound object variables. In *Proc. of National Computer Security Conference*, pages 56–67, 1987.
39. B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proc. of IEEE Symposium on Security and Privacy*, pages 184–200, 2001.
40. A. D. Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *Proc. of IEEE Computer Security Foundations Workshop*, June 2002.
41. R. P. Reitman. *Information flow in parallel programs: an axiomatic approach.* PhD thesis, Cornell University, 1978.
42. P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach.* Addison-Wesley, 2001.
43. A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. of International Conference on Perspectives of System Informatics*, *LNCS 2244*, pages 227–241, July 2001.
44. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. of IEEE Computer Security Foundations Workshop*, July 2000.
45. 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.
46. G. Smith. A new type system for secure information flow. In *Proc. of IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.
47. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. of Symposium on Principles of Programming Languages*, 1998.
48. D. Volpano. Safety versus secrecy. In *Proc. of Symposium on Static Analysis*, *LNCS 1694*, pages 303–311, September 1999.
49. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, November 1999.
50. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
51. S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 2002. To appear.
52. S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. of ACM Symposium on Operating System Principles*, October 2001.

Appendix A

✓=**"secure"** ×=**"insecure"** $dom(cid)=dom(cid')$

program	***cid*- *enc***	***cid*- *high***
send:		
send(cid, Exp)	✓	
if Exp_h then send(cid, Exp) else skip	×	✓
if Exp_h then send(cid, Exp) else send(cid', Exp')	×	✓
if Exp_h then send(cid, Exp) else send(cid, Exp')	✓	
send(cl, Exp_h) $(dom(cl) = low)$	×	
receive:		
receive(cid, h)	✓	×
if Exp_h then receive(cid, h) else skip	×	
if Exp_h then receive(cid, h) else receive(cid', h)	×	
if Exp_h then receive(cid, h) else receive(cid, h')	✓	×
receive(cid, l)	×	
if-receive:		
if-receive(cid, h, C_1, C_2)	$C_1 \approx_L C_1$ $C_2 \approx_L C_2$	$C_1 \approx_L C_2$
if Exp_h then if-receive(cid, h, C_1, C_2) else (skip; C_3)	×	$C_1 \approx_L C_2$ $C_2 \approx_L C_3$
if-receive(cid, l, C_1, C_2)	×	
synch-send $(dom(destReady[cid]) = dom(destAck[cid]) = dom(cid)$ for $cid \in CID)$**:**		
synch-send(cid, h)	✓	×
if Exp_h then synch-send(cid, Exp) else skip	×	
if Exp_h then synch-send(cid, Exp) else synch-send(cid', Exp')	×	
if Exp_h then synch-send(cid, Exp) else synch-send(cid, Exp')	✓	×
synch-send(cl, Exp_h) $(dom(cl) = low)$	×	
if Exp_h then send(cid, Exp) else synch-send(cid, Exp)	×	
synch-receive $(dom(destReady[cid]) = dom(destAck[cid]) = dom(cid)$ for $cid \in CID)$**:**		
synch-receive(cid, h)	✓	×
if Exp_h then synch-receive(cid, h) else skip	×	
if Exp_h then synch-receive(cid, h) else synch-receive(cid', h)	×	
if Exp_h then synch-receive(cid, h) else synch-receive(cid, h')	✓	×
synch-receive(cid, l)	×	
if Exp_h then receive(cid, h) else synch-receive(cid, h)	×	

Fig. 7. Examples on communication primitives and their impact on security

Refinement of LTL Formulas for Abstract Model Checking

María del Mar Gallardo, Pedro Merino, and Ernesto Pimentel

Dpto. de Lenguajes y Ciencias de la Computación
University of Málaga, 29071 Málaga, Spain
{gallardo,pedro,ernesto}@lcc.uma.es

Abstract. A crucial problem in abstract model checking is to find a tradeoff between constructing the "best" (the smallest) abstract model, approximating a given model, and preserving as much interesting properties over the original model as possible. In this paper, we present a method for dealing with this problem based on the definition of a new abstract satisfiability relation. This new relation allows us to analyze temporal properties with different degrees of precision, by means of a refinement process. The method subsumes the classic way of abstracting properties and the dual proposal of the authors. As a consequence, maintaining the same abstract model, we directly obtain the preservation of universal properties (as in the classic method) and the refutation of existential properties (as in the dual method). We also show the utility of this method by proving that the very important notions of completeness and precision in abstract model checking may be analyzed by using the new relation. In particular, we exploit the power of model checking to simultaneously refine both the model and the properties.[1]

1 Introduction

Abstract model checking integrates the two well-known techniques of model checking [3,2] and abstract interpretation [8] to reduce the state space produced when verifying realistic and complex models, which permits the analysis of bigger systems using the available tools.

Abstract model checking involves two related activities. On the one hand, the first step is to apply abstract interpretation in order to reduce the model to be analyzed. Considering that models are represented as transition systems $M = \langle \Sigma, \rightarrow \rangle$, this task basically consists of abstracting data (states) and operations (transitions) as in the classic applications of abstract interpretation. Thus, the abstraction of states consists of defining a set of abstract states Σ^α and relating concrete states with their approximations in Σ^α. This may be done by defining Galois connections as in [19], abstraction functions as in [16,10, 7,12] or surjective homomorphisms as in [4,6]. Regarding the transitions, we find in existing literature [6,10] two complementary definitions. One of them,

[1] This research is partially supported by the CICYT projects TIC2001-2705-C03-02 and TIC99-1083-C02-01.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 395–410, 2002.
© Springer-Verlag Berlin Heidelberg 2002

$M^\alpha = \langle \Sigma^\alpha, \rightarrow^\alpha \rangle$, produces an over-approximation of M in the sense that each concrete transition in M is abstracted by one in M^α, while the other, here denoted by $M^\alpha_\exists = \langle \Sigma^\alpha, \rightarrow^\alpha_\exists \rangle$, is an under-approximation, this meaning that each abstract transition corresponds to a concrete one. Abstract transition relations $\rightarrow^\alpha$ and $\rightarrow^\alpha_\exists$ are respectively called *liberal* and *conservative* in [6] and *free* and *constrained* in [10]. An alternative method for abstracting the model is the abstraction of predicates that replaces selected predicates in the model by Boolean variables [1,22,23].

Once the model has been reduced, we must also abstract the original satisfiability relation $\models$, which evaluates temporal properties against concrete models, and define an abstract relation to reinterpret the meaning of properties against the abstract models. Given a generic temporal property f, the ideal objective of the abstraction process is "strong preservation" (that is, the preservation of both the truth and the falsehood) of the universal ($\forall f$) and existential ($\exists f$) properties between M and M^α. However, strong preservation of universal and existential properties is only possible if M and M^α are bi-similar [19], which entails a considerable constraint when the objective is to decrease the state space. Thus, it is accepted that a reasonable construction of abstract models may involve some loss of information when analyzing temporal properties.

Following the same ideas discussed above, we may consider different abstractions of $\models$, each preserving a different set of temporal formulas. For instance, the abstract satisfiability relation defined in [4,6,10] under-approximates properties in such a way that the abstract model satisfies less properties than the concrete one. We denote this relation as $\models^\alpha_c$. This definition directly produces the weak preservation of universal (resp. existential) properties from M^α (resp. $M^\alpha_\exists$) to M. Alternatively, the dual definition $\models^\alpha$ that we studied in [12] preserves the refutation of existential (resp. universal) properties in the same direction. Developing this dual approach, we complete abstract model checking with the same two objectives as standard model checking (satisfaction and refutation), and users can choose the method best suited to their personal preferences or to the formulation of property to be analyzed.

Proposals [6,10] both achieve weak preservation of universal and existential properties by integrating free and constrained abstract transition relations to construct the abstract model. Universal properties are proved using the free relation, while for existential properties, the constrained relation is used. However, the verification of properties over the abstract model may produce non-conclusive answers due to the imprecision and incompleteness of the abstract model. For instance, the non-satisfaction of a universal property over the abstract model may be produced by a *spurious trace*, that is, an abstract trace which does not correspond to any concrete one. Similar problems occur in the dual method.

Solutions for this problem follow different directions. On the one hand, the analysis of counter-examples serves to find out whether the trace is real or spurious [5,23,21]. Another possibility is to improve the abstract model by a refinement process as in [10]. In [15], authors prove that the strong preservation of properties in abstract model checking is strongly connected to the precision of the underlying abstract domain. In particular, they show how spurious counter-examples can be removed by refining the abstract domain. The theoretical basis

of this approach has been developed in a more general framework for achieving completeness in abstract interpretation [14]. In [6] the problem of the optimality in abstract model checking is studied by considering that the abstraction of the model also involves the abstraction of properties. In [9] authors present a new combination of the backward and forward analysis (techniques widely employed in abstract interpretation) for improving verification of universal safety properties.

In this paper, we present a new proposal for dealing with the problem of optimality in abstract model checking, in the context of linear time temporal logic (LTL) [20]. The idea is to define an abstract satisfiability relation $\Vdash$ combining and extending both the classic $\models_c^\alpha$ and the dual $\models^\alpha$ methods. The benefits of this relation may be summarized as follows:

- Considering a unique abstract model M^α (the over-approximated version), we simultaneously achieve the preservation of the satisfaction of universal properties and the refutation of existential ones.
- The extended relation allows us to formalize the notion of precision of the abstract model with respect to the analysis of a given property.
- In addition, using relation $\Vdash$ allows us to implicitly refine the model. The model checking tool exclusively produces the part of M^α required to analyze the property.
- The approach is suitable for refining properties depending on the actual precision of the abstract model.

The new satisfiability relation proposed is efficiently implementable. In fact, we have constructed a tool based on this proposal [13,24].

The paper is organized as follows. Section 2 is devoted to the construction of the abstract model. Section 3 presents and relates the classic method for abstracting properties and the over-approximated one, providing a framework with a common terminology. Section 4 presents the extended relation and its application for the verification of temporal formulas. In addition, we study its extension to include the analysis of refined formulas. Finally, in Section 5, we give conclusions and future work.

2 Abstracting Concurrent Systems

Execution of a concurrent program may be defined by means of *labelled transition systems* such as $M = (A, \Sigma, \bar{\rightarrow}, s_0)$ where A is the set of *observable atomic actions*, Σ is the set of standard *states*, $\bar{\rightarrow} \subseteq \Sigma \times A \times \Sigma$ is a labelled transition relation and s_0 is the initial state. We write $s \xrightarrow{a} s'$ for $(s, a, s') \in \bar{\rightarrow}$. A *trace* x of M is a sequence $x = t_0 \xrightarrow{a_0} t_1 \xrightarrow{a_1} \ldots$ of states, and it represents a (possibly infinite) computation from state t_0, where $a_0 a_1 \ldots$ is the sequence of actions executed. We denote by x^j the suffix path $t_j \xrightarrow{a_j} t_{j+1} \xrightarrow{a_{j+1}} \ldots$ of a trace $x = t_0 \xrightarrow{a_0} \ldots$. A *full-trace* $x = t_0 \xrightarrow{a_0} \ldots$ is an infinite trace. We assume that terminating traces have a final state which is repeated forever. The set

$\mathcal{O}(M) = \{x \mid x = s_0 \xrightarrow{a_0} \ldots \textit{ is a full} - \textit{trace}\}$ defines the trace semantics determined by the transition system M.

We may construct an abstract interpretation $M^\alpha = (A, \Sigma^\alpha, \bar{\rightarrow}_\alpha, s_0^\alpha)$ of a transition system $M = (A, \Sigma, \bar{\rightarrow}, s_0)$ by means of a triple $\mathcal{I}_\alpha = (\Sigma, (\Sigma^\alpha, \leq^\alpha), \alpha)$, where $\alpha : 2^\Sigma \rightarrow \Sigma^\alpha$ is the lower adjoint (the abstraction function) of a Galois insertion between $(2^\Sigma, \subseteq)$ and the lattice of abstract states $(\Sigma^\alpha, \leq^\alpha)$. Partial order $\leq^\alpha$ denotes the degree of precision of each abstract state, the smallest elements being the most precise ones. Function α associates each state s with its "best" approximation $\alpha(\{s\}) \in \Sigma^\alpha$ wrt $\leq^\alpha$. In the sequel, we write $\alpha(s)$ for $\alpha(\{s\})$.

Given $x = t_0 \xrightarrow{a_0} \cdots$, $\alpha(x)$ denotes the abstract trace $\alpha(t_0) \xrightarrow{a_0}_\alpha \cdots$. Abstract trace $\alpha(x)$ is the most precise approximation of x. However, note that possibly $\alpha(x)$ is not an element of $\mathcal{O}(M^\alpha)$. Finally, given $x^\alpha = t_0^\alpha \xrightarrow{a_0}_\alpha \cdots$ and $y^\alpha = r_0^\alpha \xrightarrow{a_0}_\alpha \cdots$, we write $x^\alpha \leq^\alpha y^\alpha$ when $\forall i \geq 0. t_i^\alpha \leq^\alpha r_i^\alpha$.

Definition 1. *M^α is $\mathcal{I}_\alpha-$correct wrt M, iff $\forall x \in \mathcal{O}(M)$ there exists $x^\alpha \in \mathcal{O}(M^\alpha)$ such that $\alpha(x) \leq^\alpha x^\alpha$.*

$\mathcal{I}_\alpha-$correctness means that the abstract system over-approximates the concrete one in two directions. First, the abstract trace x^α approximating x in $\mathcal{O}(M)$ is usually less precise than $\alpha(x)$, which means that abstract traces usually lose information with respect to concrete ones. Imprecision of abstract traces directly affects the analysis of properties. In addition, abstract models are often incomplete, that is, the construction of abstract transition systems may produce the so-called *"spurious"* traces, which can lead to obtaining false answers when analyzing temporal properties.

In the sequel, we always assume that M^α is $\mathcal{I}_\alpha-$correct wrt the original model M.

3 Abstracting Temporal Logic: Relating Two Dual Approaches

Kripke structures are used to evaluate temporal formulas against models. In this section, we summarize the classic approach and the method based on over-approximation for abstracting Kripke structures. We also discuss the main preservation results that may be deduced from these two dual definitions. In order to easily integrate both approaches, we consider *weak* Kripke structures where the *negation* $\neg$ is not dealt with as a connective, but as a way of constructing an atomic proposition.

3.1 Temporal Logic

Given $Prop$ a set of propositions, we construct the set $\mathcal{P} = Prop \cup \neg Prop$, where $\neg Prop = \{\neg p : p \in Prop\}$. Let $\mathcal{F}$ be the set of LTL temporal formulas built inductively using the elements of $\mathcal{P}$, the standard Boolean operators, except $\neg$,

and the *temporal operators*: *next* "$\bigcirc$", *always* "$\Box$", *eventually* "$\Diamond$", and *until* "U".

A labelled transition system $M = (A, \Sigma, \bar{\rightarrow}, s_0)$ may be extended to a *weak Kripke* structure $\mathcal{K}_\mathcal{P} = \langle M, \tau \rangle$ where $\tau : \Sigma \rightarrow 2^\mathcal{P}$ is a function that assigns truth values to the propositions of $\mathcal{P}$ in each state.

$\mathcal{K}_\mathcal{P} = \langle M, \tau \rangle$ is a *Kripke* structure iff $\forall s \in \Sigma, \forall p \in Prop$ the *Principle of Excluded Middle* (PEM) (i. e., $p \in \tau(s) \vee \neg p \in \tau(s)$), and the *Principle of Non-Contradiction* (PNC) (i. e., $p \not\in \tau(s) \vee \neg p \not\in \tau(s)$) hold.

Note that negated propositions are explicitly included in $\mathcal{P}$. In the sequel, $p \in \mathcal{P}$ denotes both positive and negative atomic propositions. In addition, we assume that $\neg\neg p = p$. Structure $\mathcal{K}_\mathcal{P} = \langle M, \tau \rangle$ defines an interpretation of atomic propositions, which may be rewritten as a satisfiability relation $\models^\tau$ over traces $x = t_0 \stackrel{a_0}{\rightarrow} \ldots$ using the initial state t_0 to evaluate the propositions as:

$$x \models^\tau p \;\Leftrightarrow\; p \in \tau(t_0)$$

Now we can extend $\models^\tau$ to temporal formulas by defining the meaning of the temporal operators as follows.

Definition 2. *Let $\mathcal{K}_\mathcal{P} = \langle M, \tau \rangle$ be a weak Kripke structure. Given a trace x, and $f, g \in \mathcal{F}$, we define relation $\models^\tau$ inductively as follows:*

$$\begin{array}{lcl}
x \models^\tau f \vee g & \Leftrightarrow & x \models^\tau f \textit{ or } x \models^\tau g. \\
x \models^\tau f \wedge g & \Leftrightarrow & x \models^\tau f \textit{ and } x \models^\tau g. \\
x \models^\tau f \rightarrow g & \Leftrightarrow & x \models^\tau f \textit{ implies } x \models^\tau g. \\
x \models^\tau \bigcirc f & \Leftrightarrow & x^1 \models^\tau f. \\
x \models^\tau \Box f & \Leftrightarrow & \forall k \geq 0. x^k \models^\tau f. \\
x \models^\tau \Diamond f & \Leftrightarrow & \exists k \geq 0. x^k \models^\tau f. \\
x \models^\tau f\ U g & \Leftrightarrow & \exists k \geq 0. (x^k \models^\tau g \textit{ and } \forall j < k. (x^j \models^\tau f)).
\end{array}$$

Finally, we extend $\models^\tau$ as follows.

1. Universal formulas: $M \models^\tau \forall f$ *iff* $\forall x \in \mathcal{O}(M). x \models^\tau f$.
2. Existential formulas: $M \models^\tau \exists f$ *iff* $\exists x \in \mathcal{O}(M). x \models^\tau f$.[2]

The set $\mathcal{F}$ includes all temporal formulas in negation normal form, that is, negations only appear in atomic propositions. Note that we have not included a rule defining the satisfaction of a negated formula. Instead, we treat the evaluation of negated propositions independently of their corresponding non-negated ones. The reason for this will be explained below.

In the following three sections, we present and compare the two dual methods for abstracting the satisfiability relation $\models$. In what follows, in order to simplify notation, we will write $\models$, $\models^\alpha$ instead of $\models^\tau$, $\models^{\tau^\alpha}$, respectively. Besides, to differentiate between the two different ways of abstracting the satisfiability relation considered in the paper, we will use $\models_c^\alpha$ when we refer to the classic relation.

[2] Note that symbols $\forall$ and $\exists$ are not LTL connectives. We use them to simplify the notation.

3.2 The Classic Method

Let $\mathcal{K}_{\mathcal{P}} = \langle M, \tau \rangle$ a weak Kripke structure. The classic abstraction of $\mathcal{K}_{\mathcal{P}}$, $\mathcal{K}^{\alpha}_{\mathcal{P}} = \langle M^{\alpha}, \tau^{\alpha}_{c} \rangle$, is given by

$$\tau^{\alpha}_{c}(s^{\alpha}) = \bigcap_{\{\alpha(s) \leq^{\alpha} s^{\alpha}\}} \tau(s) \qquad (Under_c)$$

Note that the codomain of both τ and τ^{α}_{c} coincide. Usually, $\mathcal{K}_{\mathcal{P}}$ is a Kripke structure, that is, it satisfies the conditions PNC and PEM. However, note that the way of defining τ^{α}_{c} makes it possible that for a given abstract state s^{α} and a proposition $p \in \mathcal{P}$, neither $p \in \tau^{\alpha}_{c}(s^{\alpha})$ nor $\neg p \in \tau^{\alpha}_{c}(s^{\alpha})$ occur. This is why we skipped the rule for negated propositions from Definition 2. Condition $Under_c$ has a number of interesting properties.

1. τ^{α}_{c} under-approximates τ. Given $s^{\alpha} \in \Sigma^{\alpha}$,

$$\forall s.(\alpha(s) \leq^{\alpha} s^{\alpha} \Rightarrow \tau(s) \supseteq \tau^{\alpha}_{c}(s^{\alpha})) \qquad (A_c)$$

2. τ^{α}_{c} is the biggest set verifying the condition (A_c). Given $s^{\alpha} \in \Sigma^{\alpha}$, and $\mathcal{Q} \subseteq \mathcal{P}$,

$$\textit{if } \forall s.(\alpha(s) \leq^{\alpha} s^{\alpha} \Rightarrow \tau(s) \supseteq \mathcal{Q}) \textit{ then } \tau^{\alpha}_{c}(s^{\alpha}) \supseteq \mathcal{Q} \qquad (C_c)$$

3. τ^{α}_{c} is monotonic decreasing. Given $s^{\alpha}_{1}, s^{\alpha}_{2} \in \Sigma^{\alpha}$,

$$\textit{if } s^{\alpha}_{1} \leq^{\alpha} s^{\alpha}_{2} \textit{ and } \exists s.\alpha(s) \leq^{\alpha} s^{\alpha}_{1} \textit{ then } \tau^{\alpha}_{c}(s^{\alpha}_{1}) \supseteq \tau^{\alpha}_{c}(s^{\alpha}_{2}) \qquad (M_c)$$

4. A_c and C_c univocally determine τ^{α}_{c},

$$Under_c \iff A_c \wedge C_c \qquad (E_c)$$

5. The extension to abstract traces preserves the satisfiability relation from the abstract to the concrete model, that is, given $f \in \mathcal{F}$

$$\alpha(x) \leq^{\alpha} x^{\alpha} \;\Rightarrow\; (x^{\alpha} \models^{\alpha}_{c} f \Rightarrow x \models f) \qquad (P_c)$$

Condition P_c assures the weak conservation of universal properties from the abstract to the concrete model as stated in the following theorem.

Theorem 1. *Given $f \in \mathcal{F}$, if $M^{\alpha} \models^{\alpha}_{c} \forall f$ then $M \models \forall f$.*

This theorem is equivalent to the assertion $M \not\models \forall f \Rightarrow M^{\alpha} \not\models^{\alpha}_{c} \forall f$. However, assuming that $\neg f$ is in negation normal form, since $M^{\alpha} \not\models^{\alpha}_{c} \forall f \Rightarrow M^{\alpha} \models^{\alpha}_{c} \exists \neg f$ holds only if $\mathcal{K}^{\alpha}_{\mathcal{P}}$ is a Kripke structure, the theorem cannot be used to refute temporal formulas. In the following section, we will see that refutation is obtained by changing under-approximation by over-approximation.

3.3 The Over-Approximation Method

Considering the abstract weak Kripke structure $\mathcal{K}^{\alpha}_{\mathcal{P}} = \langle M^{\alpha}, \tau^{\alpha} \rangle$, where condition $Under_c$ is substituted by

$$\tau^{\alpha}(s^{\alpha}) = \bigcup_{\{\alpha(s) \leq^{\alpha} s^{\alpha}\}} \tau(s) \qquad (Over)$$

we obtain a dual approach for abstracting the satisfiability relation $\models$. Note that with this definition, it is possible that for a given abstract state s^{α} and a proposition $p \in \mathcal{P}$, either $p \in \tau^{\alpha}(s^{\alpha})$ and $\neg p \in \tau^{\alpha}(s^{\alpha})$ occur.

The following properties of τ^{α} are dual to the ones given in the preceding section:

1. τ^{α} over-approximates τ. Given $s^{\alpha} \in \Sigma^{\alpha}$,

$$\forall s.(\alpha(s) \leq^{\alpha} s^{\alpha} \Rightarrow \tau(s) \subseteq \tau^{\alpha}(s^{\alpha})) \qquad (A)$$

2. τ^{α} is the smallest set verifying the condition (A). Given $s^{\alpha} \in \Sigma^{\alpha}$, and $\mathcal{Q} \subseteq \mathcal{P}$,

$$if \forall s.(\alpha(s) \leq^{\alpha} s^{\alpha} \Rightarrow \tau(s) \subseteq \mathcal{Q}) \ then \ \tau^{\alpha}(s^{\alpha}) \subseteq \mathcal{Q} \qquad (C)$$

3. τ^{α} is monotonic increasing. Given $s^{\alpha}_1, s^{\alpha}_2 \in \Sigma^{\alpha}$,

$$if \ s^{\alpha}_1 \leq^{\alpha} s^{\alpha}_2 \ then \ \tau^{\alpha}(s^{\alpha}_1) \subseteq \tau^{\alpha}(s^{\alpha}_2) \qquad (M)$$

4. A and C univocally determine τ^{α},

$$Over \iff A \wedge C \qquad (E)$$

5. The extension to abstract traces preserves the satisfiability relation from the concrete to the abstract model, that is, given $f \in \mathcal{F}$

$$\alpha(x) \leq^{\alpha} x^{\alpha} \ \Rightarrow \ (x \models f \Rightarrow x^{\alpha} \models^{\alpha} f) \qquad (P)$$

Condition P assures the refutation of existential properties from the abstract to the concrete model.

Theorem 2. *Given $f \in \mathcal{F}$, if $M^{\alpha} \not\models^{\alpha} \exists f$ then $M \not\models \exists f$.*

As before, note that the previous theorem cannot be used to analyze the satisfaction of universal properties, since in general, assuming that formula $\neg f$ is in negation normal form, $M^{\alpha} \models^{\alpha} \exists f \ /\!\!\Rightarrow M^{\alpha} \not\models^{\alpha} \forall \neg f$.

Example 1. Consider the model $M = (A, Int, \bar{\rightarrow}, s_0)$, containing an integer variable a and the abstraction function $\alpha_{np} : 2^{Int} \rightarrow NegPos^{\alpha}$ which transforms sets of integer values into one of the values of the lattice $NegPos^{\alpha} = \{\bot, neg, nopos, zero, noneg, pos, \top\}$, in the natural way. Figure 1 shows the partial order defined over $NegPos^{\alpha}$. Assume that $\mathcal{K}_{\mathcal{P}} = \langle M, \tau \rangle$ is a Kripke structure and consider propositions $a > 0$, $a \leq 0 \in \mathcal{P}$.

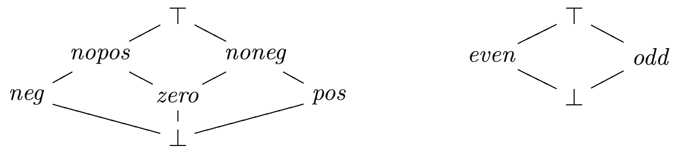


Fig. 1. Partial Orders over $NegPos^\alpha$ and $EvenOdd^\alpha$

Following the classic method, $a > 0 \in \tau_c^\alpha(a^\alpha) \;\Leftrightarrow\; a^\alpha = pos$ and $a \leq 0 \in \tau_c^\alpha(a^\alpha) \;\Leftrightarrow\; a^\alpha \in \{neg, zero, nopos\}$. However, for the over-approximation method, proposition $a > 0 \in \tau^\alpha(a^\alpha)$ iff $a^\alpha \in \{pos, noneg, \top\}$, and $a \leq 0 \notin \tau^\alpha(a^\alpha)$ iff $a^\alpha = pos$.

Therefore, checking $M^\alpha \models_c^\alpha \forall\Box(a > 0)$ directly proves that variable a has always a positive value, and dually, checking $M^\alpha \not\models^\alpha \exists\Diamond(a \leq 0)$ proves that a never takes a non-positive value. Note that both methods may prove the same property but using dual approaches.

3.4 Relating the Classic and the Over-Approximation Methods

The original model $\mathcal{K}_\mathcal{P}$ used to be a Kripke structure, that is, conditions PEM and PNC hold. However, we may weaken these conditions by only imposing *precision* on the propositions to be analyzed. We say that $p \in \mathcal{P}$ is *precise* in the original structure $\mathcal{K}_\mathcal{P} = \langle M, \tau \rangle$ iff $\forall s \in \Sigma$,

$$p \in \tau(s) \Leftrightarrow \neg p \notin \tau(s) \qquad (Prec_p)$$

Condition $Prec_p$ is reasonable because we cannot obtain precision over the abstract model, if we do not have precision over the original one. In the rest of the paper, we always assume that the propositions to be analyzed are precise wrt the original structure. This means that $\mathcal{K}_\mathcal{P}$ may be a *weak* Kripke structure obtained from a previous process of abstraction.

Under these conditions, we obtain the following result relating the classic and the over-approximated methods.

Proposition 1. *Given $p \in \mathcal{P}$ and $f \in \mathcal{F}$,*

$$(a)\; \forall s^\alpha \in \Sigma^\alpha.(\neg p \notin \tau^\alpha(s^\alpha) \quad \Leftrightarrow p \in \tau_c^\alpha(s^\alpha))$$
$$(b)\; \forall x^\alpha \in \mathcal{O}(M^\alpha).(x^\alpha \not\models^\alpha \neg f \Leftrightarrow x^\alpha \models_c^\alpha f)$$

Result (a) clearly shows the dual effect of the loss of information about a proposition p on both methods. Result (b) extends this effect to traces and temporal formulas.

Proposition 2. *Given $f \in \mathcal{F}$,*

$$(a)\; M^\alpha \models_c^\alpha \forall f \Rightarrow M^\alpha \models^\alpha \forall f$$
$$(b)\; M^\alpha \not\models^\alpha \exists f \Rightarrow M^\alpha \not\models_c^\alpha \exists f$$
$$(c)\; M^\alpha \models_c^\alpha \forall f \Leftrightarrow M^\alpha \not\models^\alpha \exists\neg f$$

Results (a) and (b) hold due to the fact that the classic method under-approximates properties and the dual method over-approximates them. Result (c) is more interesting since it means that the two methods succeed under the same conditions. In other words, the complementary result of Theorem 1 is Theorem 2, and the opposite.

In addition, Proposition 2(c) also shows that both methods may be used for proving or refuting a given property. However, this is not true in general because it is not always possible/efficient to construct/verify the negation normal form of a temporal formula. For instance, the negation normal form of $\neg(pUq)$ makes use of the release operator "$\bigvee$" which is not implemented in most model checkers. Thus, in practice, we cannot always use the classic method for refuting properties, and dually the over-approximated method cannot be always used to prove universal properties.

From the other point of view, we can also say that the classic method cannot give a conclusive answer regarding the satisfaction of a universal property, iff the over-approximation method cannot refute its negation. The reasons for the failure are complementary in each method. Classic method fails because condition PEM does not hold, and the over-approximation method fails because PNC does not hold. This situation occurs when the abstract model is incomplete or when the abstract traces are imprecise with respect to the formula to be proved. Imprecision is usually obtained during the construction of the abstract models, and in general it cannot be avoided.

Example 2. In the context of Example 1, assume that $inc1, dec1 \in A$ are two actions that respectively increase and decrease the value of variable a by 1. The trace $x = 1 \overset{inc1}{\rightarrow} 2 \overset{inc1}{\rightarrow} 3 \overset{dec1}{\rightarrow} 2 \rightarrow \cdots$ may be abstracted by $x^\alpha = pos \overset{inc1}{\rightarrow} pos \overset{inc1}{\rightarrow} pos \overset{dec1}{\rightarrow} noneg \rightarrow \cdots$. Observe that $\alpha(x) \leq^\alpha x^\alpha$, but since the fourth state is approximated by *noneg* and not by *pos*, x^α is more imprecise than $\alpha(x)$. This is because the decrement by 1 of a positive value may produce the value *zero* or another positive number. Note that, in this case, imprecision can not be avoided. Thus, if $x^\alpha \in \mathcal{O}(M^\alpha)$ then $M^\alpha \not\models^\alpha_c \forall\Box(a > 0)$ and $M^\alpha \models^\alpha \exists\Diamond(a \leq 0)$.

However, the abstraction of states should be chosen in such a way that the propositions to be analyzed are not imprecise from the beginning. The notion of *consistency* introduces this idea.

Proposition $p \in \mathcal{P}$ is *consistent* wrt τ and $\mathcal{I}_\alpha$ iff $\forall s \in \Sigma$,

$$p \in \tau^\alpha(\alpha(s)) \Rightarrow p \in \tau(s)$$

Example 3. Consider again the Kripke structure used in Example 1 and the abstraction function $\alpha_{eo} : 2^{Int} \rightarrow EvenOdd^\alpha$ which abstracts integer values into the values of $EvenOdd^\alpha = \{\bot, even, odd, \top\}$ in the natural way. Figure 1 illustrates the partial order defined over this set. Proposition $a = 1$ is not *consistent* wrt τ and $\mathcal{I}_{\alpha_{eo}}$, since $a = 1 \in \tau^\alpha(\alpha_{eo}(3))$ but $a = 1 \notin \tau(3)$. In contrast, it is easy to prove that proposition $a \ mod\ 2 \neq 0$ is *consistent*.

When all atomic propositions in $\mathcal{P}$ are consistent, we obtain a definition equivalent to the one given in [19]. The following proposition clarifies this equivalence.

Proposition 3.

$$p \in \mathcal{P} \textit{ is consistent wrt } \tau \textit{ and } \mathcal{I}_\alpha \textit{ iff } \forall s.(p \in \tau_c^\alpha(\alpha(s)) \vee \neg p \in \tau_c^\alpha(\alpha(s)))$$

Proof. Necessary condition. Given $s \in \Sigma$, let us assume that there exist two states $s_1, s_2 \in \Sigma$ such that $\alpha(s_1) \leq^\alpha \alpha(s)$, $\alpha(s_2) \leq^\alpha \alpha(s)$, and $p \in \tau(s_1), \neg p \in \tau(s_2)$. By Definition *Over*, this implies that $p, \neg p \in \tau^\alpha(\alpha(s))$, and applying the hypothesis, we have that $p, \neg p \in \tau(s)$. But this is not possible if p is precise in the original structure. Therefore, either $\forall s'.\alpha(s') \leq^\alpha \alpha(s), p \in \tau(s')$ or $\forall s'.\alpha(s') \leq^\alpha \alpha(s), \neg p \in \tau(s')$. That is, $p \in \tau_c^\alpha(\alpha(s)) \vee \neg p \in \tau_c^\alpha(\alpha(s))$.

Sufficient condition. Assume that $p \in \tau^\alpha(\alpha(s))$. Then, by definition, there exists $s_1 \in \Sigma$ such that $\alpha(s_1) \leq^\alpha \alpha(s)$ and $p \in \tau(s_1)$. Using the hypothesis, this implies that $\forall s'.\alpha(s') \leq^\alpha \alpha(s)$, $p \in \tau(s')$. In particular, $p \in \tau(s)$. □

This means the consistency of a proposition may be equivalently proved using the classic and the over-approximation methods. Proposition 3 also proves that only consistent propositions may be analyzed by the classic method. However, the over-approximation method may additionally use inconsistent propositions as studied in [12].

As commented in the introduction, imprecision and incompleteness in abstract model checking is usually analyzed by refining the abstract model or analyzing the counter-examples obtained. In Section 4.2, we present an alternative proposal. Since the over-approximation method introduces inconsistencies and the classic method eliminates them, we propose to refine formulas by progressively eliminating inconsistencies when using the over-approximation method or by introducing them when using the classic one. This refinement process makes use of an extended satisfiability relation which is developed below.

4 Refinement of Temporal Formulas

From the preceding discussion, and imitating the usual way of verifying in (standard) model checking, we may argue that the ideal approach for abstract model checking is the use of the classic method for proving the satisfaction of some key properties, and the over-approximation method to discard critical errors. Furthermore, in this section, we propose a common framework to integrate, and even to extend, the capabilities of these two basic methods. Note that the advantage of this combination is that the abstract model is not modified. This section also contains some methodological guidelines for the practical use of this framework.

Definition 3. *Consider the sets* $\mathcal{P}^\alpha = \{p^\alpha : p \in \mathcal{P}\}$ *and* $\overline{\mathcal{P}} = \mathcal{P} \cup \mathcal{P}^\alpha$. *We construct the weak Kripke structure* $\mathcal{K}^\alpha_{\overline{\mathcal{P}}} = \langle M^\alpha, \overline{\tau}^\alpha \rangle$, *where* $\overline{\tau}^\alpha : \Sigma^\alpha \to \overline{\mathcal{P}}$ *is defined as:*

$$p \in \overline{\tau}^\alpha(s^\alpha) \quad \Leftrightarrow \neg p \ / \notin \tau^\alpha(s^\alpha)$$
$$p^\alpha \in \overline{\tau}^\alpha(s^\alpha) \Leftrightarrow p \in \tau^\alpha(s^\alpha)$$

Let $\Vdash$ denote the satisfiability relation $\models^{\overline{\tau}^\alpha}$. The set $\mathcal{P}^\alpha$ contains the abstract versions of the elements of $\mathcal{P}$. Intuitively, $x^\alpha \Vdash p$ means that the abstract satisfaction of p does not involve loss of information, that is, it means that the abstraction process has not affected proposition p. In addition, $x^\alpha \Vdash p^\alpha$ means that the abstract satisfaction of p "may" involve loss of information. Clearly, $x^\alpha \Vdash p \Rightarrow x^\alpha \Vdash p^\alpha$, but the opposite may fail if the abstraction process has modified the meaning of p.

Let $\overline{\mathcal{F}}$ denote the set of temporal formulas which can be constructed using the atomic propositions of $\overline{\mathcal{P}}$. We may construct the abstract versions of the temporal formulas in $\mathcal{F}$ by inductively substituting each sub-formula by its abstraction as follows: $(f \vee g)^\alpha = f^\alpha \vee g^\alpha$, $(f \wedge g)^\alpha = f^\alpha \wedge g^\alpha$,$(f \to g)^\alpha = f^\alpha \to g^\alpha$, $(\Diamond f)^\alpha = \Diamond f^\alpha$, $(\Box f)^\alpha = \Box f^\alpha$, $(\bigcirc f)^\alpha = \bigcirc f^\alpha$, and $(fUg)^\alpha = f^\alpha U g^\alpha$.

We may extend $\Vdash$ to temporal formulas as in Definition 2.

Proposition 1 allows us to assert that $x^\alpha \Vdash p \Leftrightarrow x^\alpha \models^\alpha_c p$ which means that relation $\models^\alpha_c$ may be expressed in terms of $\Vdash$. Extending this assertion and the one given by Definition 3 to temporal formulas, we obtain that

$$x^\alpha \Vdash f^\alpha \Leftrightarrow x^\alpha \models^\alpha f$$
$$x^\alpha \Vdash f \Leftrightarrow x^\alpha \models^\alpha_c f$$

which mean that $\Vdash$ may be used to model both the classic relation $\models^\alpha_c$, and $\models^\alpha$.

4.1 Imprecision and Incompleteness

The extended relation $\Vdash$ may help us know, and even modify, the degree of precision when analyzing temporal properties. In addition, this technique may serve to refine the model by using model checking power to discard traces which are not interesting for any reason (for instance, because it is known that they are "spurious".) The next definition states the notion of precision.

Definition 4. *We say that formula* $f \in \mathcal{F}$ does not lose precision *wrt* $\mathcal{O}(M^\alpha)$ *iff* $\forall x^\alpha \in \mathcal{O}(M^\alpha)$, *if* $x^\alpha \models^\alpha f$ *then* $x^\alpha \not\models^\alpha \neg f$.

Note that the notion of precision is associated with a given property. Thus, even though abstract models are, in general, imprecise, they may be useful for analyzing some properties precisely. By Proposition 1, Definition 4 could have also been given as $x^\alpha \models^\alpha_c f$ instead of $x^\alpha \not\models^\alpha \neg f$ which allows us to formulate, in the following proposition, the notion of precision in terms of $\Vdash$. This means that we may use the model checker to automatically analyze the precision of the abstract model with respect to a given property. The proposition also proves that, when the formula f is precise wrt the abstract model, the two methods may be used to respectively check the satisfaction of $\forall f$ and the refutation of $\exists f$.

Proposition 4. *Given $f \in \mathcal{F}$, if $M^\alpha \Vdash \forall(f^\alpha \to f)$ then*

$$(a)\ M^\alpha \models_c^\alpha \forall f \Leftrightarrow M^\alpha \models^\alpha \forall f$$
$$(b)\ M^\alpha \not\models_c^\alpha \exists f \Leftrightarrow M^\alpha \not\models^\alpha \exists f$$

Proof. (a) By Proposition 2(a), $M^\alpha \models_c^\alpha \forall f \Rightarrow M^\alpha \models^\alpha \forall f$. Conversely, if $x^\alpha \models^\alpha f$ then $x^\alpha \Vdash f^\alpha$, and by hypothesis, this means that $x^\alpha \Vdash f$. Therefore, $M^\alpha \models^\alpha \forall f \Rightarrow M^\alpha \models_c^\alpha \forall f$. (b) By Proposition 2(b), $M^\alpha \not\models^\alpha \exists f \Rightarrow M^\alpha \not\models_c^\alpha \exists f$. Conversely, if $x^\alpha \models^\alpha f$ then $x^\alpha \Vdash f^\alpha$. By hypothesis, this means that $x^\alpha \Vdash f$ or, equivalently, $x^\alpha \models_c^\alpha f$ which is not possible. Therefore, $M^\alpha \not\models_c^\alpha \exists f \Rightarrow M^\alpha \not\models^\alpha \exists f$. □

Proposition 5. *Let $f, g \in \mathcal{F}$.*

$$(a)\ M \models \forall g \text{ and } M^\alpha \Vdash \forall(g^\alpha \to f) \Rightarrow M \models \forall f$$
$$(b)\ M \not\models \exists g \text{ and } M^\alpha \Vdash \forall(f^\alpha \to g) \Rightarrow M \not\models \exists f$$

Proof. Given $x \in \mathcal{O}(M)$, by the $\mathcal{I}_\alpha$−correctness, $x^\alpha \in \mathcal{O}(M^\alpha)$ exists such that $\alpha(x) \leq^\alpha x^\alpha$. (a) By hypothesis, $x \models g$ which implies (Condition P) that $x^\alpha \models^\alpha g$, that is, $x^\alpha \Vdash g^\alpha$. Using the hypothesis this means that $x^\alpha \Vdash f$, or equivalently, $x^\alpha \models_c^\alpha f$. Finally, by Condition P_c, we obtain that $x \models f$. Thus, (a) holds. (b) By hypothesis, $x \not\models g$ and $x^\alpha \Vdash f^\alpha \to g$. Therefore, using condition P_c, $x^\alpha \not\Vdash f^\alpha$, that is, $x^\alpha \not\models^\alpha f$. And, by condition P, we obtain that $x \not\models f$. □

The utility of Proposition 5 is clear. We may utilize previously checked formulas to refine the model. Thus, for instance, if we know that the original model satisfies a given property, we may discard all the abstract traces which do not satisfy it. As before, the crucial point here is that the task of refinement is automatically realized by the model checker.

A very important application of this proposition is the elimination of the so-called non-progress cycles. Abstraction may add abstract traces that do not correspond to any original one. For instance, it is usual that the abstraction process transforms terminating loops into non-deterministic loops including non-progress cycles. Many liveness properties may be violated by abstract traces containing these cycles. But these traces are really "spurious", thus, if we avoid them, we augment the precision of our analysis. Assume that *progress* represents the absence of non-progress cycles from a given state and that it is known that $M \models \forall\Box progress$; then, in order to prove that a formula $f \in \mathcal{F}$ holds, we may discard the abstract traces that do not satisfy $\Box progress$ by using the formula $(\Box progress^\alpha) \to f$ to prove f. In addition, usually property $progress$[3] is not affected by the abstraction process, that is, it coincides with $progress^\alpha$.

4.2 Intermediate Precision

Relation $\Vdash$ induces a partial order relation $\Rightarrow$ over the set of formulas $\overline{\mathcal{F}}$ as $f_1 \Rightarrow f_2 \Leftrightarrow \forall x^\alpha \in \mathcal{O}(M^\alpha).x^\alpha \Vdash f_1 \Rightarrow x^\alpha \Vdash f_2$. Clearly, it holds that for

[3] Non-progress cycles are usually analyzed without inspecting the whole state space.

all $f \in \mathcal{F}, f \Rightarrow f^\alpha$ and this relation means that the classic method gives an abstract interpretation of f more precise than the one provided by the over-approximation method.

Let us assume that we may construct an intermediate Kripke structure $\mathcal{K}^i_{\mathcal{P}} = \langle M^\alpha, \tau^\alpha_i \rangle$, such that for all $s^\alpha \in \Sigma^\alpha$, $\tau^\alpha_c(s^\alpha) \subseteq \tau^\alpha_i(s^\alpha) \subseteq \tau^\alpha(s^\alpha)$. Using τ^α_i and Definition 2, we may define an intermediate satisfiability relation $\models^\alpha_i$. Now we extend the relation $\Vdash$ to include the precision given by the abstract Kripke structure $\mathcal{K}^i_{\mathcal{P}}$ as follows.

Consider the sets $\mathcal{P}^i = \{p^i : p \in \mathcal{P}\}$ and $\overline{\mathcal{P}} = \mathcal{P} \cup \mathcal{P}^\alpha \cup \mathcal{P}^i$. We construct the weak Kripke structure $\mathcal{K}^\alpha_{\overline{\mathcal{P}}} = \langle M^\alpha, \overline{\tau}^\alpha \rangle$, where $\overline{\tau}^\alpha : \Sigma^\alpha \to \overline{\mathcal{P}}$ is defined as:

$$\begin{array}{l} p \in \overline{\tau}^\alpha(s^\alpha) \Leftrightarrow p \in \tau^\alpha_c(s^\alpha), \\ p^i \in \overline{\tau}^\alpha(s^\alpha) \Leftrightarrow p \in \tau^\alpha_i(s^\alpha) \\ p^\alpha \in \overline{\tau}^\alpha(s^\alpha) \Leftrightarrow p \in \tau^\alpha(s^\alpha) \end{array}$$

Now consider the extension of $\overline{\mathcal{F}}$ including the abstract propositions with intermediate precision. In addition, given a temporal formula f, we may construct the formula f^i by inductively substituting each sub-formula by its intermediate version as was done previously for formulas f^α. The extension of $\Vdash$ including the new formulas satisfies that

$$\begin{array}{l} x^\alpha \Vdash f^\alpha \Leftrightarrow x^\alpha \models^\alpha f \\ x^\alpha \Vdash f^i \Leftrightarrow x^\alpha \models^\alpha_i f \\ x^\alpha \Vdash f \Leftrightarrow x^\alpha \models^\alpha_c f \end{array}$$

In addition, for all $f \in \mathcal{F}$, $f \Rightarrow f^i \Rightarrow f^\alpha$.

Example 4. Following Example 1, assume that we define $a > 0$ with a intermediate precision τ^α_i between the precisions given by τ^α_c and τ^α as follows: $\tau^\alpha_i(a^\alpha) = \tau^\alpha(a^\alpha), \forall a^\alpha \not\exists$, and $\tau^\alpha_i(\top) = \tau^\alpha(\top) - \{a > 0\}$. Using the notation introduced above, we have that $a > 0 \Rightarrow (a > 0)^i \Rightarrow (a > 0)^\alpha$.

4.3 Methodological Guidelines

As in standard model checking, there are two complementary approaches for ensuring the correctness of a model. In the "satisfaction-oriented method", the user must specify the desired property to be held over the model. Denote this property with $f \in \mathcal{F}$. If $M^\alpha \Vdash \forall f$, then by Theorem 1 property f holds on all traces of M. However, if $M^\alpha \not\Vdash \forall f$, the user may continue the analysis with the formula $\forall f^\alpha$. If $M^\alpha \not\Vdash \forall f^\alpha$, the generous way of defining f^α makes an error on M very probable, except for spurious traces. However, if $M^\alpha \Vdash \forall f^\alpha$, the user knows that "from the abstract point of view", the model satisfies $\forall f$, although this information may be too imprecise. Thus, the user may refine it with an intermediate formula f^i, verifying $f \Rightarrow f^i \Rightarrow f^\alpha$. If $M^\alpha \Vdash \forall f^i$, then the user knows that property f holds on all traces of M until the precision defined by f^i, and this may be enough for her/him. Note that we could incrementally refine the formula until the desired precision is achieved. The probability of obtaining

a real error when $M^\alpha \not\Vdash \forall f^i$ decreases when the precision of f^i increases. As the set of possible intermediate p^i for a given p is finite, part of this task can be done with a semi-automatic tool.

Example 5. In the context of Examples 1 and 4, consider the property "If a is null and it is incremented by 1, then it takes a positive value", which may be formulated as $f \equiv a = 0 \wedge inc \rightarrow \Diamond(a > 0)$. If $M^\alpha \not\Vdash \forall f$, then the user knows that there exists a trace where a has been incremented by 1 but it never takes the abstract value *pos*. Assume that $M^\alpha \Vdash \forall f^\alpha$, but since this result is too imprecise we construct the more precise version of f^α, $f^i \equiv (a = 0) \wedge inc \rightarrow \Diamond(a > 0)^i$, where $(a > 0)^i$ represents $(a > 0)^\alpha \wedge (a \neq \top)$. Note that $f \Rightarrow f^i \Rightarrow f^\alpha$. Now, if $M^\alpha \Vdash \forall f^i$, we know that after incrementing by 1, variable a has taken some of the abstract values *pos* or *noneg*, but it has never taken the value *neg*. This may be due to the imprecision when abstracting operations as illustrated in Example 2. If the increment *inc* was implemented by adding 2 and then subtracting 1, the result may produce the imprecise value *noneg*. In this case, the information given by $M^\alpha \Vdash \forall f^i$ may be sufficient for the user.

The "refutation-oriented" method could be alternatively used. In this case, the user must specify the erroneous behavior as a temporal formula to be refused. Denote this formula with f. If $M^\alpha \not\Vdash \exists f^\alpha$, by Theorem 2, user knows that no trace of M matches f. However, if $M^\alpha \Vdash \exists f^\alpha$, the user may continue the analysis with the formula $\exists f$. Since the classic method carries out a very precise analysis, if $M^\alpha \Vdash \exists f$, then it is very probable, except for spurious traces, that an erroneous trace exists in M. Otherwise, if $M^\alpha \not\Vdash \exists f$, the user could improve the analysis with a less precise formula f^i such that $f \Rightarrow f^i \Rightarrow f^\alpha$. If $M^\alpha \not\Vdash \exists f^i$, the user knows that no trace satisfies f until the imprecision given by f^i. Formula f^i may be refined until the desired imprecision is obtained. The probability of obtaining a real error when $M^\alpha \Vdash \exists f^i$ decreases when f^i decreases.

5 Conclusions

Previous works dealing with abstract model checking focus on satisfaction of temporal formulas and treat the problems of imprecision and incompleteness by means of the model refinement [4,6,10,5,15]. In this work, we have shown that these problems may be also solved from a dual perspective, that is, we may extract information from an incomplete abstract model by refining the formulas to be analyzed against it. In this discussion, the key point is the combination of the classic and the over-approximated methods for abstracting properties.

Note that we have distinguished between incompleteness and imprecision, although, as it has been extensively studied in [14], both notions reflect the same situation from the abstract interpretation perspective: the abstract model does not reproduce the concrete one exactly. However, from the abstract model checking point of view, these two types of incompleteness have to be differently treated since, while imprecise traces cannot be removed from the abstract model (they approximate some concrete traces), it would be desirable to eliminate spurious traces. Of course, an imprecise trace may be also transformed into a

set of more precise abstract traces, possibly including some spurious traces. But this transformation involves a modification of the abstract model.

The results presented in this paper are complementary (dual) to the above mentioned in three key aspects: a) the integration of the refutation of temporal formulas in abstract model checking; b) the study of imprecision and incompleteness problems by means of the refinement of formulas; c) the combination of the classic and the over-approximation methods in a uniform framework. We think that all these proposals are compatible with the other works for improving abstract model checking. In particular, if no information is obtained by the refinement of formulas, we can switch to model refinement or, even, in the worst case, to modifying the abstraction.

One future work is to develop a similar framework using the under-approximated model ($M^{\alpha}_{\exists}$) (extending the works in [10,6]) for debugging the model. Note that in this case the counter-examples found are real.

Of course, the ideal aim is to integrate all existing techniques in the same tool. In this direction, we have already implemented the proposals presented in the paper. Our current tool, αSpin[13], extends the model checker Spin [17, 18] with syntactic transformation for PROMELA (its modelling language) and LTL (its version of temporal logic). The transformation of PROMELA allows us to obtain a new model that exhibits the behaviours of M^{α} over-approximating M (presented in [11]). The partial/full transformation of LTL produces a new formula which is verified against M^{α}. Using standard Spin verification with these transformed specifications corresponds to implementing the satisfaction relations $\models^{\alpha}$, $\models^{\alpha}_{c}$ and $\Vdash$. The transformation cost is negligible, whereas the experiences with real examples show considerable benefits for verification. The tool can be downloaded free from [24].

References

1. S. Bensalem, Y. Lakhnech, S. Owre, Computing abstractions of infinite state systems compositionally and automatically. In *Computer-Aided Verification CAV'98*, LNCS-1427, pp. 319–331, (1998).
2. E.M. Clarke, O. Grumberg, D. Peled. *Model Checking* (The MIT Press, 2000).
3. E.M. Clarke, E. A. Emerson, A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. on Programming Languages and Systems*, **8**(2) (1986) 244–263.
4. E.M. Clarke, O. Grumberg, D.E. Long. Model Checking and Abstraction. *ACM Transaction on Languages and Systems*, **16**(5) (1994) 1512–1245.
5. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification CAV'00*, LNCS-1855, pp. 154–169, (2000).
6. R. Cleveland, P. Iyer, D. Yankelevich. Optimality in Abstractions of Model Checking. In *Proceedings of Static Analysis Symposium* LNCS-983, pp. 51–63, (1995)
7. J. Corbett, M. Dwyer, J. Hatcliff, L. Shawn, C. Pasareanu, H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *Proc. 22nd Int. Conf. On Software Engineering*, pp. 439–448, (2000).

8. P. Cousot, R. Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Record of the 4th ACM Symp. on Princ. of Prog. Languages*, pp. 238-252, (1977).
9. P. Cousot, R. Cousot. Refining Model Checking by Abstract Interpretation. *Automated Software Engineering* **6** (1999) 69–95.
10. D. Dams, R. Gerth, O. Grumberg. Abstract Interpretation of Reactive Systems. *ACM Trans. on Programming Languages and Systems*, **19**(2) (1997) 253–291.
11. M.M. Gallardo, P. Merino. A Framework for Automatic Construction of Abstract PROMELA Models. In *Theoretical and Practical Aspects of SPIN Model Checking*, LNCS-1680, pp. 184–199, (1999).
12. M.M. Gallardo, P. Merino, E. Pimentel. Verifying Abstract LTL Properties on Concurrent Systems. In *Proc. of the 6th World Conference on Integrated Design & Process Technology*, (2002).
13. M. M. Gallardo, J. Martínez, P. Merino, E. Pimentel. αSPIN: Extending SPIN with Abstraction. In *Proc. of the 9th International SPIN Workshop on Model Checking of Software*, LNCS 2318, pp. 254–258, (2002).
14. R. Giacobazzi, F. Ranzato, F. Scozzari. Making abstract interpretation complete. In *Journal of ACM*, **47**(2) (2000), 361–416.
15. R. Giacobazzi, E. Quintarelli. Incompleteness, Counterexamples and Refinement in Abstract Model-Checking. In *The 8th International Static Analysis Symposium SAS'01*, LNCS 2126, pp. 356–373, (2001).
16. S. Graf. Verification of a distributed Cache Memory by using abstractions. In *Computer Aided Verification CAV'94*, LNCS-818, pp. 207-219, (1994).
17. G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
18. G.J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering* **23**(5) (1997) 279–295.
19. C. Loiseaux , S. Graf, J. Sifakis, A. Boujjani, S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design* **6** (1995) 1–35.
20. Z. Manna , A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - Specification*. Springer-Verlag, New York, (1992).
21. C.S. Pasareanu, M.B. Dwyer, W. Visser. Finding Feasible Counter-examples when Model Checking Abstracted Java Programs. In *Proc. of 7th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001*, LNCS-2031, pp. 284–298, (2001).
22. H. Saïdi, N. Shankar. Abstract and model check while you prove. In *Computer-Aided Verification CAV'99* LNCS-1633, pp. 443–454, (1999).
23. H. Saïdi. Model Checking guided abstraction and analysis. In *Seventh International Static Analysis Symposium (SAS'00)*, LNCS-1824, pp. 377–396, (2000).
24. αSpin project. University of Málaga. `http://www.lcc.uma.es/~gisum/fmse/tools` .

Making Abstract Model Checking Strongly Preserving

Francesco Ranzato[1] and Francesco Tapparo[1,2]

[1] Dipartimento di Matematica Pura ed Applicata
Università di Padova
Via Belzoni 7, 35131 Padova, Italy
franz@math.unipd.it

[2] Dipartimento di Matematica
Università di Milano, Milano, Italy
tapparo@math.unipd.it

Abstract. Usually, abstract model checking is not strongly preserving: it may well exist a temporal specification which is not valid on the abstract model but which is instead satisfied by the concrete model. Starting from the standard notion of bisimulation, we introduce a notion of completeness for abstract models: completeness together with a so-called partitioning property for abstract models implies strong preservation for the past μ-calculus. Within a rigorous abstract interpretation framework, we show that the least refinement of a given abstract model, for a suitable ordering on abstract models, which is complete and partitioning always exists, and it can be constructively characterized as a greatest fixpoint. This provides a systematic methodology for minimally refining an abstract model checking in order to get strong preservation.

1 Introduction

Abstract model checking is one practical way to deal with the well-known state explosion problem of model checking [3,6]. The essence of the abstract model checking approach to system verification is well known and established: the checking of the correctness specification is performed over an abstraction of the system model, called abstract model, which abstracts away from some properties of the system model [4,5]. Thus, abstract model checking basically consists in applying an abstract interpretation-based approach to the model of the system to check. An abstract state space is considered, where an abstract state approximates some selected properties of a concrete state of the system. An abstract transition relation is defined between abstract states and this gives rise to the abstract model. The check of the temporal specification is performed on the abstract model, and a correctness result, also called preservation result, ensures that if a specification holds on the abstract model then it is also holds on the concrete model. The abstract model checking approach has been very successful, and allowed to verify systems with a tremendous number of states (for instance, a ALU circuit with 10^{1300} reachable states was verified already in early works

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 411–427, 2002.
© Springer-Verlag Berlin Heidelberg 2002

[5]), since suitably chosen abstractions can lead to great reductions of the size of the model to verify. The foundations and principles of abstract model checking have been studied and improved in many directions. For instance, it has been shown that an abstract model checking approach can be used for verifying infinite-state systems [1,11], since the basic idea is substantially the same: an infinite-state model is approximated and then reduced to an abstract model with a finite number of states. Let us point out that abstract model checking can be rigorously specified within standard Cousot and Cousot's [7,8] abstract interpretation theory [9,10,16,17], and this paper follows this principle as well.

The design of abstract model checking always comes together with a preservation result. *Strong preservation* results are highly desirable: a formula of a suitable specification language is valid on the abstract model if and only if it is valid on the concrete model. Thus, strong preservation allows to draw consequences from negative answers on the abstract side: if a formula is not valid abstractly then it is not valid on the concrete model. It should be clear that strong preservation is related to the idea of *complete abstract interpretation* [8, 15]. Actually, this has been recently noted by Giacobazzi and Quintarelli [13], who studied and related completeness in asbtract interpretation, strongly preserving abstract model checking and the so-called Clarke et al.'s [2] spurious counterexamples of abstract model checking.

The idea of enhancing the precision of abstract interpretations by refining the underlying abstract domains dates back to the early works by Cousot and Cousot [8], and evolved to the systematic design of abstract interpretations by abstract domain refinements [12,14]. In particular, Giacobazzi et al. [15] thorougly investigated the abstract domain refinement making abstract interpretations complete. They showed that completeness for an abstract interpretation is a property which does not depend on the abstract semantic operations but on the underlying abstract domains only. This opened up the question of making abstract interpretations complete by least refinements of abstract domains. It turns out that this least refinement exist and can be constructively characterized as a fixed point solution of abstract domain equations.

On the model checking side, the idea of systemically refining abstract model checking to enhance its precision is due to Clarke et al. [2]. They introduced the notion of spurious counterexample in abstract model checking: given a temporal formula φ, an abstract execution trace π is a spurious conunterexample for φ if π does not satisfy φ whereas there exists a concrete execution trace whose abstraction is π but which is not a counterexample for φ. Then, they devised a methodology for refining an abstract model checking w.r.t. a given temporal specification φ by using the spurious counterexamples provided by the abstract model checker on φ. Giacobazzi and Quintarelli [13] casted spurious counterexamples as a lack of completeness, in the abstract interpretation sense, for the abstract model checking. Then, by applying the results in [15] for minimally making abstract interpretations complete, they designed a method for systematically refining abstract model checking in order to eliminate spurious counterexamples.

This paper follows the idea of applying systematic refinement operators to abstract model checking in order "to gain precision". However, here the per-

spective is different from that of [2,13]: our main goal is that of systematically refining abstract model checking in order to get strong preservation for all the formulae of a suitable temporal specification language.

Firstly, we single out a new (to the best of our knowledge) notion of *complete abstract model*: if (Q, R) and (A, S) are, respectively, the concrete and abstract transition systems and $\rho \subseteq Q \times A$ is the abstraction relation of simulation then (A, S) is complete whenever $R^{-1}\rho = \rho S^{-1}$. Hence, such relation of completeness is, in a sense, dual to the standard bisimulation relation. Strong preservation is related to completeness by the following result: completeness for a so-called *partitioning* abstraction relation implies strong preservation for the past μ-calculus. The partitioning property for the abstraction relation ρ comes from the work of Loiseaux et al. [16] and means that ρ induces a partition of the abstract state space A. Thus, our goal is that of "minimally making an abstract model checking complete and partitioning". Of course, this requires a notion of ordering for the abstract models of a given concrete transition system. This is done by isolating the notions of partitioning and complete closure operators on sets of system states and by defining a bijective correspondence between these closure operators and abstract models which preserves the partitioning and completeness properties. This allows us to move our problem to the standard abstract interpretation framework using closure operators. Closures are a well-known and useful tool for specifying abstract domains in abstract interpretation theory [8], and they are particularly helpful when reasoning on abstract domains independently from the representation of their objects. In this framework, we constructively solve the problem of characterizing the least complete and partitioning refinement of the closure μ_ρ on $\wp(Q)$ associated to the abstraction relation $\rho \subseteq Q \times A$: this is the, necessarily unique, greatest, w.r.t. the standard pointwise ordering, closure μ_ρ^* which is a refinement of μ and which is both partitioning and complete. Analogously to [15], this problem is formulated as a fixpoint equation between abstract domains and hence it is constructively solved. Then, we come back to abstract models by associating to this least refinement μ_ρ^* an abstract model (A^*, R^*): this refined abstract model is related to the concrete model (Q, R) by an abstraction relation ρ^* which is partitioning and induces completeness. We reached our goal: we have systematically designed a strongly preserving abstract model (A^*, R^*) which is a minimal refinement of (A, S) in the following sense: any other complete and partitioning abstract model (B, T) of (Q, R) which is a refinement of (A, S) turns out to be a simulation of (A^*, R^*).

2 Preliminaries

2.1 Abstract Interpretation Basics

The structure $\langle \mathrm{uco}(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top, \lambda x.x \rangle$ denotes the complete lattice of all (upper) closure operators on a complete lattice $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$, where $\mu \sqsubseteq \eta$ iff $\forall x \in C.\ \mu(x) \leq \eta(x)$. Throughout the paper, for any $\mu \in \mathrm{uco}(C)$, we follow a standard notation by denoting the image $\mu(C)$ simply by μ itself: This does not give rise to ambiguity, since one can readily distinguish the use of μ as function or set according to the context. Let us recall that (i) each closure $\mu \in \mathrm{uco}(C)$ is

uniquely determined by the set of its fixpoints, which coincides with its image, i.e. $\mu = \{x \in C \mid \mu(x) = x\}$, (ii) $\mu \sqsubseteq \eta$ iff $\eta \subseteq \mu$, and (iii) a subset $X \subseteq C$ is the set of fixpoints of a closure iff $X = \mathcal{M}(X) \stackrel{\text{def}}{=} \{\wedge Y \mid Y \subseteq X\}$ ($\mathcal{M}(X)$ is called the Moore-closure of X; note that $\top = \wedge\varnothing \in \mathcal{M}(X)$). Let us also recall that a closure is additive (i.e., preserves arbitrary lub's, empty set included) iff its set of fixpoints is closed by lub's.

Within the standard Cousot and Cousot framework, abstract domains can be equivalently specified either by Galois connections/insertions (GCs/GIs) or by closure operators [8]. In the first case, concrete and abstract domains C and A — for simplicity, let C and A be complete lattices — are related by a pair of adjoint maps $\alpha : C \to A$ and $\gamma : A \to C$, compactly denoted by (α, C, A, γ), and therefore C and A may consist of objects having different representations. In the second case, instead, an abstract domain is specified as a closure operator on the concrete domain C. Thus, the closure operator approach is particularly convenient when reasoning about properties of abstract domains independently from the representation of their objects. Given a concrete domain C, we will identify $\mathrm{uco}(C)$ with the so-called complete lattice of abstract interpretations of C (cf. [7,8]). The ordering on $\mathrm{uco}(C)$ corresponds precisely to the standard order used in abstract interpretation to compare abstract domains with regard to their precision: A_1 is more precise (or concrete) than A_2 iff $A_1 \sqsubseteq A_2$ in $\mathrm{uco}(C)$. Thus, lub's $\sqcup$ and glb's $\sqcap$ on $\mathrm{uco}(C)$ give, respectively, the most precise abstraction and the most abstract concretization of a family of abstract domains.

Complete Abstract Interpretations. Let us briefly recall the basic notions concerning completeness in abstract interpretation. Let $f : C \to C$ be a monotone or antitone concrete semantic function[1] occurring in some complex semantic specification, and let $f^\sharp : A \to A$ be a corresponding abstract function on the abstract domain A. Then, $\langle A, f^\sharp\rangle$ is a sound abstract interpretation — or $f^\sharp$ is a correct approximation of f relatively to A — when $\forall c \in C.\, \alpha(f(c)) \leq_A f^\sharp(\alpha(c))$. On the other hand, $\langle A, f^\sharp\rangle$ is complete when equality holds, i.e. $\alpha \circ f = f^\sharp \circ \alpha$. Thus, completeness means that abstract computations accumulate no loss of information.

Any abstract domain A induces the so-called canonical best correct approximation $f^A : A \to A$ of $f : C \to C$, defined by $f^A \stackrel{\text{def}}{=} \alpha \circ f \circ \gamma$. This terminology is justified by the fact that any $f^\sharp : A \to A$ is a correct approximation of f iff $f^A \sqsubseteq f^\sharp$. Consequently, any abstract domain always induces an (automatically) sound abstract interpretation. However, not all abstract domains induce a complete abstract interpretation. It turns out that whenever a complete abstract function exists then this actually is the best correct approximation. This therefore means that completeness for an abstract function is a property which depends on the underlying abstract domain only. Consequently, for abstract domains specified by closure operators, an abstract domain $\mu \in \mathrm{uco}(C)$ is defined to be complete for f if $\mu \circ f = \mu \circ f \circ \mu$ (see [15] for more details).

[1] For simplicity, we consider unary functions with the same domain and co-domain; the extension to the general case is conceptually straightforward.

We call abstract domain refinement any operator performing an action of refinement on abstract domains, with respect to their standard ordering of precision $\sqsubseteq$ [8,14]. In particular, the concepts of shell and shell refinement with respect to a given property of abstract domains, namely a given subclass of $\mathrm{uco}(C)$, are as follows.

Definition 2.1. *Let C be a complete lattice and $\mathcal{P} \subseteq \mathrm{uco}(C)$ be an abstract domain property. Let $\mu, \sigma \in \mathrm{uco}(C)$. Then, σ is the $\mathcal{P}$-shell of μ if:*

1. $\sigma \in \mathcal{P}$,
2. $\sigma \sqsubseteq \mu$,
3. *if $\eta \in \mathcal{P}$ and $\eta \sqsubseteq \mu$ then $\eta \sqsubseteq \sigma$.*

An operator $\mathcal{F} : \mathrm{uco}(C) \to \mathrm{uco}(C)$ is called the $\mathcal{P}$-shell refinement if for any $\mu \in \mathrm{uco}(C)$, $\mathcal{F}(\mu)$ is the $\mathcal{P}$-shell of μ. □

Thus, the $\mathcal{P}$-shell of an abstract domain A, if this exists, is the most abstract among the domains which satisfy $\mathcal{P}$ and are more precise than A. Complete shells with respect to a concrete interpretation $f : C \to C$ are particularly important. They have been studied by Giacobazzi et al. in [15], where the authors characterized through fixpoints the complete shell refinement under the weak hypothesis that f is a continuous function.

2.2 Abstract Model Checking

We follow a general approach to abstract model checking [16]. If $R \subseteq X \times Y$ and $S \subseteq Y \times Z$ are relations then $R^{-1} \subseteq Y \times X$ denotes the reverse relation of R and $RS \subseteq X \times Z$ denotes the composition of the two relations. We deal with generic transition systems (Q, R), where Q is any (finite or infinite) set of system states and $R \subseteq Q \times Q$ is the transition relation. Let us recall the well-known notions of simulation and bisimulation between transition systems. Let $\mathcal{T} = (Q, R)$ and $\mathcal{T}' = (A, S)$ be transition systems and $\rho \subseteq Q \times A$ be a *relation of abstraction* (also called *abstraction relation*). Then (i) $\mathcal{T}'$ is a *ρ-abstraction* of $\mathcal{T}$, or $\mathcal{T}$ ρ-simulates $\mathcal{T}'$, denoted by $\mathcal{T} \sqsubseteq_\rho \mathcal{T}'$, if $R^{-1}\rho \subseteq \rho S^{-1}$; (ii) $\mathcal{T}$ *ρ-bisimulates* $\mathcal{T}'$, denoted by $\mathcal{T} \simeq_\rho \mathcal{T}'$, if $\mathcal{T} \sqsubseteq_\rho \mathcal{T}'$ and $\mathcal{T}' \sqsubseteq_{\rho^{-1}} \mathcal{T}$. $\mathcal{T}$ and $\mathcal{T}'$ are called, respectively, the concrete and abstract model.

Definition 2.2. *Let (Q, R) be a transition system. An abstraction (of Q) is simply a triple $\langle Q, \rho, A\rangle$ where A is any set and $\rho \subseteq Q \times A$.* □

An abstraction $\langle Q, \rho, A\rangle$ induces the usual pre/post transformers:

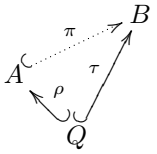
- $\mathrm{pre}[\rho] \stackrel{\mathrm{def}}{=} \lambda Y.\{a \in Q \mid \exists b \in Y.\ a\rho b\} : \wp(A) \to \wp(Q)$
- $\widetilde{\mathrm{pre}}[\rho] \stackrel{\mathrm{def}}{=} \lambda Y.\{a \in Q \mid \forall b \in A.(a\rho b \Rightarrow b \in Y)\} : \wp(A) \to \wp(Q)$
- $\mathrm{post}[\rho] \stackrel{\mathrm{def}}{=} \lambda X.\{b \in A \mid \exists a \in X.\ a\rho b\} : \wp(Q) \to \wp(A)$
- $\widetilde{\mathrm{post}}[\rho] \stackrel{\mathrm{def}}{=} \lambda X.\{b \in A \mid \forall a \in Q.(a\rho b \Rightarrow a \in X\} : \wp(Q) \to \wp(A)$

Moreover, $\mathrm{dom}(\rho) \stackrel{\mathrm{def}}{=} \{q \in Q \mid \exists a \in A.\ q\rho a\}$ and $\mathrm{img}(\rho) \stackrel{\mathrm{def}}{=} \{a \in A \mid \exists q \in Q.\ q\rho a\}$. The relation ρ is *total* when $\mathrm{dom}(\rho) = Q$ and *surjective* when $\mathrm{img}(\rho) = A$.

Let $(Q, R) \sqsubseteq_\rho (A, S)$ and $(Q, R) \sqsubseteq_\tau (B, T)$. Simulation takes into account the transition relations. Instead, we need a way to compare the underlying abstractions of Q only.

Definition 2.3. The abstraction $\langle Q, \rho, A\rangle$ is more precise than $\langle Q, \tau, B\rangle$, denoted by $\langle Q, \rho, A\rangle \leq \langle Q, \tau, B\rangle$, iff there exists $\pi \subseteq A \times B$ such that $\tau = \rho\pi$. □

This general definition formalizes the following diagram and therefore it is the right one.



The following key notion of partitioning relation is taken from [16, Section 4].

Definition 2.4. Let $\langle Q, \rho, A\rangle$ be an abstraction. The relation ρ is called *partitioning* when $\rho = \rho\rho^{-1}\rho$. □

Hence, ρ is partitioning whenever any two states of Q which are abstracted by ρ to a common abstract state actually are abstracted by ρ exactly to the same set of abstract states. The terminology "partitioning" is justified by the following fact.

Lemma 2.5. *ρ is partitioning iff $\{\mathrm{post}[\rho](\{q\}) \mid q \in Q\}$ is a partition of A.*

Abstractions of Q can be related to closure operators on $\wp(Q)$ as follows.

Definition 2.6.

(i) Given an abstraction $\langle Q, \rho, A\rangle$, $\mu_\rho \stackrel{\mathrm{def}}{=} \widetilde{\mathrm{pre}}[\rho] \circ \mathrm{post}[\rho] \in \mathrm{uco}(\langle \wp(Q), \subseteq\rangle)$ is the closure operator associated to $\langle Q, \rho, A\rangle$.

(ii) Let Q be a set and $\mu \in \mathrm{uco}(\wp(Q))$. Then, the abstraction $\langle Q, \rho_\mu, A_\mu\rangle$ associated to μ is defined as follows: $A_\mu \stackrel{\mathrm{def}}{=} \{\mu(\{q\}) \in \wp(Q) \mid q \in Q\}$; $\langle q_1, \mu(\{q_2\})\rangle \in \rho_\mu \stackrel{\mathrm{def}}{\Leftrightarrow} \mu(\{q_1\}) = \mu(\{q_2\})$. □

The fact that $\widetilde{\mathrm{pre}}[\rho] \circ \mathrm{post}[\rho] \in \mathrm{uco}(\wp(Q))$ is a consequence of the well-known fact that $(\mathrm{post}[\rho], \wp(Q), \wp(A), \widetilde{\mathrm{pre}}[\rho])$ is a Galois connection (see e.g. [16, Section 2]). Actually, the correspondences of Definition 2.6 give rise to a bijection between total, surjective and partitioning abstractions and closure operators on $\wp(Q)$, as stated by the following result.

Lemma 2.7. *If Q is a set and $\mu \in \mathrm{uco}(\wp(Q))$, then ρ_μ is total, surjective and partitioning, and $\mu_{\rho_\mu} = \mu$. On the other hand, if $\langle Q, \rho, A\rangle$ is a total, surjective and partitioning abstraction then $\langle Q, \rho, A\rangle$ is isomorphic to $\langle Q, \rho_{\mu_\rho}, A_{\mu_\rho}\rangle$ (i.e., they are the same relation up to bijections).*

The adequacy of Definitions 2.3 and 2.6 is shown by the following result: it turns out that the preordering on abstractions is equivalent to the standard ordering between the associated closures.

Lemma 2.8. *Let* $\langle Q, \rho, A\rangle$ *and* $\langle Q, \tau, B\rangle$ *be abstractions. Then,* $\langle Q, \rho, A\rangle \leq \langle Q, \tau, B\rangle$ *iff* $\mu_\rho \sqsubseteq_{\mathrm{uco}(\wp(Q))} \mu_\tau$.

3 Complete Abstract Models

Let us consider the bisimulation property:

$$(Q, R) \simeq_\rho (A, S) \text{ iff } R^{-1}\rho \subseteq \rho S^{-1} \text{ and } S^{-1}\rho^{-1} \subseteq \rho^{-1}R^{-1}.$$

The condition of ρ-abstraction $R^{-1}\rho \subseteq \rho S^{-1}$ means that for all a, b, c there exists some d such that the following diagram commutes:

$$\begin{tikzcd} a \arrow[r, "R"] \arrow[d, "\rho"] & b \arrow[d, dotted, "\rho"] \\ c \arrow[r, dotted, "S"] & d \end{tikzcd}$$

On the other hand, the condition of ρ-bisimulation $S^{-1}\rho^{-1} \subseteq \rho^{-1}R^{-1}$ is equivalent to $\rho S \subseteq R\rho$, and this latter means that for all a, c, d there exists some b such that the following diagram commutes:

$$\begin{tikzcd} a \arrow[r, dotted, "R"] \arrow[d, "\rho"] & b \arrow[d, dotted, "\rho"] \\ c \arrow[r, "S"] & d \end{tikzcd}$$

While the relation of ρ-abstraction provides the standard order of comparison between transition systems, on the other side it is natural to consider the following diagram dual to the bisimulation diagram above:

$$\begin{tikzcd} a \arrow[r, dotted, "R"] \arrow[d, dotted, "\rho"] & b \arrow[d, "\rho"] \\ c \arrow[r, "S"] & d \end{tikzcd}$$

Thus, this diagram corresponds to the inclusion $\rho S^{-1} \subseteq R^{-1}\rho$. This leads us to the following definition of completeness between transition systems.

Definition 3.1. Let $\mathcal{T} = (Q, R)$ and $\mathcal{T}' = (A, S)$ be transition systems and $\rho \subseteq Q \times A$. Then, $\mathcal{T}'$ is a *complete ρ-abstraction* of $\mathcal{T}$, denoted by $\mathcal{T} \preccurlyeq_\rho \mathcal{T}'$, if $R^{-1}\rho = \rho S^{-1}$. □

The following simple example shows that completeness and bisimulation are orthogonal notions.

Example 3.2. Let us consider the transition systems $(\mathbb{Z}, \mathrm{succ})$ and $(A, \mathrm{succ}^\sharp)$, where $\langle m, n\rangle \in \mathrm{succ} \overset{\mathrm{def}}{\Leftrightarrow} n = m+1$, $A \overset{\mathrm{def}}{=} \{\mathrm{ev}, \mathrm{od}\}$ and $\mathrm{succ}^\sharp \overset{\mathrm{def}}{=} \{\langle \mathrm{ev}, \mathrm{od}\rangle, \langle \mathrm{od}, \mathrm{ev}\rangle\}$. Let $\rho \subseteq \mathbb{Z} \times A$ be the function mapping any integer to its parity. Then, it is easily seen that $(\mathbb{Z}, \mathrm{succ}) \simeq_\rho (A, \mathrm{succ}^\sharp)$ and $(\mathbb{Z}, \mathrm{succ}) \preccurlyeq_\rho (A, \mathrm{succ}^\sharp)$. Instead, we have that $(\mathbb{N}, \mathrm{succ}) \simeq_\rho (A, \mathrm{succ}^\sharp)$ but $(\mathbb{N}, \mathrm{succ}) \not\preccurlyeq_\rho (A, \mathrm{succ}^\sharp)$. Finally, it turns out that $(\mathbb{Z} \smallsetminus \mathbb{N}, \mathrm{succ}) \preccurlyeq_\rho (A, \mathrm{succ}^\sharp)$ but $(\mathbb{Z} \smallsetminus \mathbb{N}, \mathrm{succ}) \not\simeq_\rho (A, \mathrm{succ}^\sharp)$. □

Completeness for an abstract model w.r.t. a given concrete model, of course, depends on the abstract transition relation. However, when the abstraction relation is partitioning, it turns out that all the possible complete abstract models are bisimilar, as stated by the following result.

Theorem 3.3. *Let $\mathcal{T} = (Q, R)$ be a transition system and $\rho \subseteq Q \times A$ be surjective and partitioning. If $\mathcal{T} \preccurlyeq_\rho \mathcal{T}_1 = (A, S_1)$ and $\mathcal{T} \preccurlyeq_\rho \mathcal{T}_2 = (A, S_2)$ then $\mathcal{T}_1 \simeq_{\rho^{-1}\rho} \mathcal{T}_2$.*

It is then natural to give the following existential notion of completeness for mere state abstractions of concrete transition systems, i.e., which takes into account $\langle Q, \rho, A\rangle$ and R only, and does not depend on a specific abstract transition relation.

Definition 3.4. Let (Q, R) be a transition system. The abstraction $\langle Q, \rho, A\rangle$ is *complete* for (Q, R) if there exists $S \subseteq A \times A$ such that $(Q, R) \preccurlyeq_\rho (A, S)$. □

Example 3.5. Consider the Example 3.2. Then, as shown by Example 3.2, the abstraction $\langle \mathbb{Z}, \rho, \{\mathrm{ev}, \mathrm{od}\}\rangle$ is complete for the concrete model $(\mathbb{Z}, \mathrm{succ})$. On the other hand, let $B \overset{\mathrm{def}}{=} \{\bullet, \circ\}$ and consider the abstraction $\langle \mathbb{Z}, \zeta, B\rangle$, where $\zeta \overset{\mathrm{def}}{=} \{\langle 1, \bullet\rangle, \langle 2, \bullet\rangle\} \cup \{\langle z, \circ\rangle \mid z \in \mathbb{Z} \smallsetminus \{1, 2\}\}$. Assume that $S \subseteq B \times B$ is an abstract transition relation such that $(\mathbb{Z}, \mathrm{succ}) \sqsubseteq_\zeta (B, S)$. Then, we have that $\langle \bullet, \circ\rangle \in S$: in fact, $\langle 2, \bullet\rangle \in \zeta$ and $\langle 2, 3\rangle \in \mathrm{succ}$ and therefore, by simulation, there exists $x \in B$ such that $\langle 3, x\rangle \in \zeta$ and $\langle \bullet, x\rangle \in S$; thus, $\langle 3, x\rangle \in \zeta$ implies $x = \circ$, and therefore $\langle \bullet, \circ\rangle \in S$. Similarly, one can show that S must be the greatest relation $B \times B$, and this transition relation does not give rise to completeness. Thus, this means that the abstraction $\langle \mathbb{Z}, \zeta, B\rangle$ is not complete for $(\mathbb{Z}, \mathrm{succ})$. □

It turns out that if an abstraction is complete and partitioning then the abstract transition relation $\rho^{-1}R\rho$ induces a complete abstract model.

Theorem 3.6. *Let $\langle Q, \rho, A\rangle$ be a complete abstraction of (Q, R), where ρ is total and partitioning. Then, $(Q, R) \preccurlyeq_\rho (A, \rho^{-1}R\rho)$.*

$(A, \rho^{-1}R\rho)$ is called the *canonical complete abstract model* induced by a complete abstraction $\langle Q, \rho, A\rangle$.

Example 3.7. Let us go on with Example 3.5. We observed that the abstraction $\langle \mathbb{Z}, \rho, \{\mathrm{ev}, \mathrm{od}\}\rangle$ is complete for the concrete model $(\mathbb{Z}, \mathrm{succ})$ by considering the abstract transition relation $\mathrm{succ}^\sharp$. Actually, let us point out that $\rho^{-1}\,\mathrm{succ}\,\rho = \{\langle \mathrm{ev}, \mathrm{od}\rangle, \langle \mathrm{od}, \mathrm{ev}\rangle\} = \mathrm{succ}^\sharp$, i.e., $(\{\mathrm{ev}, \mathrm{od}\}, \mathrm{succ}^\sharp)$ is the canonical complete abstract model induced by the abstraction ρ. □

4 Strong Preservation of Temporal Calculi

Preservation and strong preservation of temporal specifications in abstract model checking are well known. Let us recall these notions in our framework. Let $\mathcal{T} = (Q, R)$ and $\mathcal{A} = (A, S)$ be transition systems. Let $\mathcal{P}$ be a set of atomic propositions used in some temporal language L. Let $\mathcal{I} : \mathcal{P} \to \wp(Q)$ be an interpretation function and $\alpha : \wp(Q) \to \wp(A)$ be any function mapping "concrete properties" to "abstract properties". In particular, α induces the abstract interpretation function $\alpha \circ \mathcal{I} : \mathcal{P} \to \wp(A)$. α is called *consistent* with $\mathcal{I}$ when for all $P \in \mathcal{P}$, $\alpha(Q \smallsetminus \mathcal{I}(P)) \cap \alpha(\mathcal{I}(P)) = \varnothing$. Then, any formula φ of L is semantically interpreted as a set $|\varphi|_{\mathcal{T},\mathcal{I}}$ of concrete states for the interpretation $\mathcal{I}$ and abstractly as a set $|\varphi|_{\mathcal{A},\alpha\circ\mathcal{I}}$ of abstract states for the interpretation $\alpha \circ \mathcal{I}$. For a subset of states $Z \subseteq Q$, the abstraction map α *preserves* a formula φ on Z for $\mathcal{I}$ whenever for any $q \in Z$,

$$q \in |\varphi|_{\mathcal{T},\mathcal{I}} \;\Rightarrow\; \alpha(\{q\}) \subseteq |\varphi|_{\mathcal{A},\alpha\circ\mathcal{I}},$$

while α *strongly preserves* φ on Z when $\Leftrightarrow$ holds. Whenever the subset Z is omitted we refer to (strong) preservation on all Q.

In this paper we deal with temporal formulae expressed within a standard μ-calculus. Let $\mathcal{X}$ be a set of variables. The formulae φ of the μ-calculus are inductively defined as follows:

$$\varphi ::= \top \mid P \in \mathcal{P} \mid X \in \mathcal{X} \mid \Diamond\varphi \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \mu X.\varphi$$

where any occurrence of a propositional variable X in φ is under an even number of negations. L_μ denotes the set of μ-calculus formulae. The remaining standard connectives of the μ-calculus, namely $\bot$, $\wedge$, $\to$, ν and $\Box$, can be defined as usual by abbreviations. $\Diamond^p$ and $\Box^p$ denote, respectively, the *past* existential and universal operators. $L_\mu^{\leftarrow}$ denotes the set of formulae of the calculus obtained by replacing $\Diamond$ with $\Diamond^p$ in the above definition of the μ-calculus: this is called the past μ-calculus.

The semantics $|\varphi|_{\mathcal{T},\mathcal{I}} \in \wp(Q)$ of a closed (i.e., without free variables) formula φ is therefore defined w.r.t. a given transition system $\mathcal{T} = (Q, R)$ and an interpretation $\mathcal{I}$. This semantics is defined as usual (see e.g. [16]). For example, let us recall that $|\Diamond\varphi|_{\mathcal{T},\mathcal{I}} \stackrel{\text{def}}{=} \mathrm{pre}[R](|\varphi|_{\mathcal{T},\mathcal{I}})$ and $|\Diamond^p\varphi|_{\mathcal{T},\mathcal{I}} \stackrel{\text{def}}{=} \mathrm{post}[R](|\varphi|_{\mathcal{T},\mathcal{I}})$.

We will use the following fragments of the μ-calculus.

$\Box^p L_\mu$ (past universal fragment):

$$\varphi ::= \top \mid \bot \mid P \mid \neg P \mid X \mid \Box^p\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \mu X.\varphi \mid \nu X.\varphi$$

$\Diamond L_\mu$ (existential fragment):

$$\varphi ::= \top \mid \bot \mid P \mid \neg P \mid X \mid \Diamond\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \mu X.\varphi \mid \nu X.\varphi$$

$\Diamond L_\mu^+$ (positive existential fragment):

$$\varphi ::= \top \mid \bot \mid P \mid X \mid \Diamond\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \mu X.\varphi \mid \nu X.\varphi$$

The plain preservation result basically states that simulation implies preservation.

Theorem 4.1 ([16, Theorem 2]). *Let $\rho \subseteq Q \times A$ and $(Q, R) \sqsubseteq_\rho (A, S)$. Let $\mathcal{I} : \mathcal{P} \to \wp(Q)$ be an interpretation. Then,* $\mathrm{post}[\rho] : \wp(Q) \to \wp(A)$ *preserves the formulae of $\Diamond L_\mu^+$, and if* $\mathrm{post}[\rho]$ *is consistent with $\mathcal{I}$ then* $\mathrm{post}[\rho]$ *preserves $\Diamond L_\mu$.*

As far as strong preservation is concerned, it is well known that bisimulation implies strong preservation. In our framework, this corresponds to the following result.

Theorem 4.2 ([16, Theorem 4]). *Let $\rho \subseteq Q \times A$ be a partitioning relation and $(Q, R) \simeq_\rho (A, S)$. Then, for any interpretation $\mathcal{I} : \mathcal{P} \to \mathrm{img}(\mathrm{pre}[\rho])$,* $\mathrm{post}[\rho]$ *strongly preserves L_μ on $\bigcup \mathrm{img}(\mathrm{pre}[\rho])$.*

We investigated the impact of complete and partitioning abstract models on strong preservation. Basically, we obtained that completeness implies strong preservation for the past universal fragment of the μ-calculus. Moreover, for the canonical complete abstract model, we are able to prove strong preservation for the full past μ-calculus. This is stated by the following result.

Theorem 4.3. *Let $\rho \subseteq Q \times A$ be total and partitioning and $(Q, R) \preccurlyeq_\rho (A, S)$. Let $\mathcal{I} : \mathcal{P} \to \mathrm{img}(\mathrm{pre}[\rho])$ be an interpretation. Then:*

1. $\mathrm{post}[\rho]$ *strongly preserves $\Box^p L_\mu$ for $\mathcal{I}$ on $\bigcup \mathrm{img}(\mathrm{pre}[\rho])$.*
2. *For the canonical complete abstract model $(A, \rho^{-1} R \rho)$,* $\mathrm{post}[\rho]$ *strongly preserves $L_\mu^{\leftarrow}$ for $\mathcal{I}$ on $\bigcup \mathrm{img}(\mathrm{pre}[\rho])$.*

5 Partitioning and Complete Closures

The aim of this section is to set up a closure operator-based framework to reason about abstract models, as far as their precision is concerned. One main advantage of handling closures is that they allow to reason about properties of abstract models independently from the representation of abstract states. Of course, this setting must be equivalent to the relation-based approach introduced above. Our first result provides a characterization of the closures associated, by Definition 2.6, to (total and) partitioning abstraction relations.

Lemma 5.1. *Let Q be a set and $\mu \in \mathrm{uco}(\wp(Q))$. Then, μ is additive and $\{\mu(\{q\})\}_{q \in Q}$ is a partition of Q iff there exists a set X and a total and partitioning $\rho \subseteq Q \times X$ such that $\mu_\rho = \mu$.*

Thus, this enables us to introduce the following notion.

Definition 5.2. Let Q be a set and $\mu \in \mathrm{uco}(\wp(Q))$. Then, μ is *partitioning* if μ is additive and $\{\mu(\{q\})\}_{q \in Q}$ is a partition of Q. $\Box$

Let us now turn to completeness. The following result shows that an abstraction $\langle Q, \rho, A \rangle$ is complete, namely there exists an abstract transition relation S such that $(Q, R) \preccurlyeq_\rho (A, S)$, if and only if the associated closure μ_ρ is complete for $\mathrm{post}[R^{-1}]$ in the classical abstract interpretation sense of Section 2.1. Observe that this is somehow surprising, since completeness for relation-based abstractions (Definition 3.4) is an existential property.

Lemma 5.3. *Let (Q, R) be a transition system and $\langle Q, \rho, A\rangle$ be an abstraction. Then, $\langle Q, \rho, A\rangle$ is complete for (Q, R) iff $\mu_\rho \circ \mathrm{post}[R^{-1}] = \mu_\rho \circ \mathrm{post}[R^{-1}] \circ \mu_\rho$.*

Thus, analogously to Definiton 5.2, this characterization justifies the following notion of completeness for closures w.r.t. a concrete model.

Definition 5.4. Let (Q, R) be a transition system and $\mu \in \mathrm{uco}(\wp(Q))$. Then, μ is *complete* for (Q, R) when $\mu \circ \mathrm{post}[R^{-1}] = \mu \circ \mathrm{post}[R^{-1}] \circ \mu$. □

Consider a closure μ which is simultaneously partitioning and complete. Then, by Definition 2.6 and Lemma 2.7, μ gives rise to a partitioning abstraction $\langle Q, \rho_\mu, A_\mu\rangle$. The following key result shows that the canonical abstract model induced by ρ_μ, as defined at the end of Section 3, actually turns out to be complete.

Theorem 5.5. *Let (Q, R) be a transition system and $\mu \in \wp(Q)$ be partitioning and complete. Then, $(Q, R) \preccurlyeq_{\rho_\mu} (A_\mu, \rho_\mu^{-1} R \rho_\mu)$.*

6 Making Abstract Models Partitioning and Complete

In the previous section we set up a closure-based framework characterizing complete and partitioning abstract model checking. As recalled in Section 2.1, closures, i.e. abstract domains, can be compared w.r.t. their precision simply by their standard pointwise ordering $\sqsubseteq$. On the other hand, we have shown in Section 4 that a complete and partitioning abstract model checking is strongly preserving. Thus, following the general approach of Definition 2.1, in this section we aim at minimally making abstract models complete and partitioning by a least refinement, so that we get a strongly preserving abstract model checking. Following the terminology of Definition 2.1, we have to show that the partitioning and complete shell of any given closure on sets of concrete states actually exists. In order to accomplish this task, first we study separately partitioning shells and complete shells, and then a combination of these solutions will solve our problem.

Definition 6.1. Given any set of states Q, define $\mathbb{P} : \mathrm{uco}(\wp(Q)) \to \mathrm{uco}(\wp(Q))$ as follows:

$$\mathbb{P}(\mu) \stackrel{\mathrm{def}}{=} \lambda Y \in \wp(Q).\{x \in Q \mid \exists y \in Y.\, \mu(\{x\}) = \mu(\{y\})\}.$$ □

It is easy to show that the operator $\mathbb{P}$ is well-defined, i.e., for any $\mu \in \wp(Q)$, $\mathbb{P}(\mu)$ is a closure. Additionally, it turns out that $\mathbb{P}(\mu)$ is an additive closure. More than this can be proved. Actually, it turns out that $\mathbb{P}$ is the right operator refining any closure to its partitioning shell.

Lemma 6.2. *$\mathbb{P}$ is the partitioning shell refinement.*

Example 6.3. Consider $Q = \{1,2,3\}$ and $\mu \in \mathrm{uco}(\wp(Q))$ as given by $\mu = \{\varnothing, \{1\}, \{2\}, \{1,2,3\}\}$. Then, μ is not partitioning, since neither $\{\mu(\{q\})\}_{q\in Q} = \{\{1\},\{2\},\{1,2,3\}\}$ is a partition of Q nor μ is additive, namely closed by set union. Then, $\mathbb{P}(\mu) = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$, i.e., the least closure, turns out to be the partitioning shell of μ. □

Let us turn to complete closures.

Definition 6.4. Given (Q,R), define $\mathbb{C}_R : \mathrm{uco}(\wp(Q)) \to \mathrm{uco}(\wp(Q))$ as follows:

$$\mathbb{C}_R(\mu) \stackrel{\mathrm{def}}{=} \lambda X \in \wp(Q).\{q \in Q \mid \mu(\mathrm{post}[R^{-1}](\{q\})) \subseteq \mu(\mathrm{post}[R^{-1}](X))\}.$$

Also, for any $\mu \in \mathrm{uco}(\wp(Q))$, define $\mathcal{C}_R^\mu : \mathrm{uco}(\wp(Q)) \to \mathrm{uco}(\wp(Q))$ as follows:

$$\mathcal{C}_R^\mu(\eta) \stackrel{\mathrm{def}}{=} \mu \sqcap \mathbb{C}_R(\eta).$$ □

It is not hard to show that $\mathbb{C}_R$ is well-defined, i.e., for any $\mu \in \wp(Q)$, $\mathbb{C}_R(\mu)$ is a closure. It turns out that $\mathbb{C}_R$ allows to characterize complete closures. As a consequence, given a closure μ the greatest fixpoint of the operator $\mathcal{C}_R^\mu$ provides the complete shell of μ.

Theorem 6.5. *Let (Q,R) be a transition system. Then, for any $\eta \in \mathrm{uco}(\wp(Q))$, η is complete iff $\eta \sqsubseteq \mathbb{C}_R(\eta)$. Moreover, $\lambda\mu.\,\mathrm{gfp}(\mathcal{C}_R^\mu)$ is the complete shell refinement.*

Finally, let us combine these separate shell refinements in an operator which simultaneously minimally refines a closure for the partitioning and complete properties. The idea is simple: basically, we consider the glb, i.e. the reduced product, of the single shell refinements. In general, it can be shown that this works for any two properties. Here, we only face with our specific case of interest.

Definition 6.6. Given a transition system (Q,R), for any $\mu \in \mathrm{uco}(\wp(Q))$, define $\mathcal{F}_R^\mu : \mathrm{uco}(\wp(Q)) \to \mathrm{uco}(\wp(Q))$ as follows:

$$\mathcal{F}_R^\mu(\eta) \stackrel{\mathrm{def}}{=} \mu \sqcap \mathbb{C}_R(\eta) \sqcap \mathbb{P}(\eta).$$ □

Corollary 6.7. *Let (Q,R) be a transition system. Then, $\lambda\mu.\,\mathrm{gfp}(\mathcal{F}_R^\mu)$ is the partitioning and complete shell refinement.*

Actually, the above operator $\mathcal{F}_R^\mu$ can be simplified as follows.

Lemma 6.8. *Let (Q,R) be a transition system.*
(1) *Define $\mathbb{C}_R^s \stackrel{\mathrm{def}}{=} \mathbb{P} \circ \mathbb{C}_R$. Then, for any $\mu \in \mathrm{uco}(\wp(Q))$,*

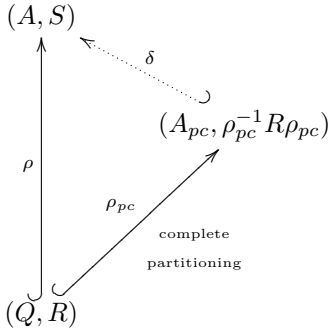
$$\mathbb{C}_R^s(\mu) = \lambda X \in \wp(Q).\{q \in Q \mid \exists x \in X.\, \mu(\mathrm{post}[R^{-1}](\{q\})) = \mu(\mathrm{post}[R^{-1}](\{x\}))\}.$$

(2) *For any $\mu \in \mathrm{uco}(\wp(Q))$, $\mathrm{gfp}(\mathcal{F}_R^\mu) = \mathrm{gfp}(\lambda\eta.\,\mu \sqcap \mathbb{C}_R^s(\eta) \sqcap \mathbb{P}(\eta))$.*

Note that for any $\mu \in \mathrm{uco}(\wp(Q))$, $\mathbb{C}_R^s(\mu)$ is a partitioning closure and therefore it is additive. Later on, we will illustrate how this methodology applies to a simple but significant example.

6.1 Refinements and Simulations

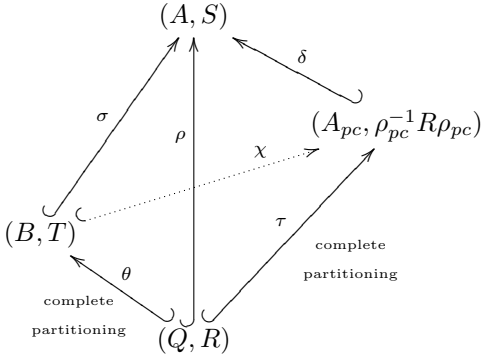
Let (Q, R) and (A, S) be transition systems such that $(Q, R) \sqsubseteq_\rho (A, S)$. Consider the associated closure $\mu_\rho \in \mathrm{uco}(\wp(Q))$, and let $\mu_\rho^{pc} \in \mathrm{uco}(\wp(Q))$ be the partitioning and complete shell of μ_ρ as given by Corollary 6.7. Let $\langle Q, \rho_{pc}, A_{pc}\rangle$ be the abstraction of Q canonically associated to μ_ρ^{pc} by Definition 2.6. Then, by Theorem 3.6, the canonical complete abstract model induced by the complete and partitioning abstraction relation ρ_{pc} is complete, i.e., $(Q, R) \preccurlyeq_{\rho_{pc}} (A_{pc}, \rho_{pc}^{-1} R \rho_{pc})$. It can be shown that the initial abstract model is an abstraction of its complete and partitioning refinement, i.e., there exists $\delta \subseteq A_{pc} \times A$ such that $(A_{pc}, \rho_{pc}^{-1} R \rho_{pc}) \sqsubseteq_\delta (A, S)$ where, additionally, $\rho = \rho_{pc}\delta$. The scenario is depicted by the following commuting diagram.



We show that $(A_{pc}, \rho_{pc}^{-1} R \rho_{pc})$ actually is the least complete and partitioning refinement of (A, S) with respect to (Q, R) for the canonical simulation preordering between transition systems. This is formalized by the following result.

Theorem 6.9. *Let (B, T) be a transition system such that: (1) $(B, T) \sqsubseteq_\sigma (A, S)$ for some $\sigma \subseteq B \times A$; (2) $(Q, R) \preccurlyeq_\theta (B, T)$ for some $\theta \subseteq Q \times B$; (3) θ is partitioning and surjective; (4) $\rho = \theta\sigma$. Then, there exists $\chi \subseteq B \times A_{pc}$ such that $(B, T) \sqsubseteq_\chi (A_{pc}, \rho_{pc}^{-1} R \rho_{pc})$.*

Thus, the closure-based methodology of refinement given in Section 6 actually is coherent with the standard simulation-based approach. Graphically, this means that the following diagram commutes.



6.2 An Example

Consider the transition systems $\mathcal{T} = (Q, R)$ and $\mathcal{A} = (A, S)$ depicted in Fig. 1. $\mathcal{A}$ is designed as the abstract model where all the concrete states except the state 5 collapse to a single abstract state. Dotted arrows from states of $\mathcal{T}$ to states of $\mathcal{A}$ formally define the abstraction relation $\rho \subseteq Q \times A$. Observe that ρ is partitioning (e.g. use Lemma 2.5).

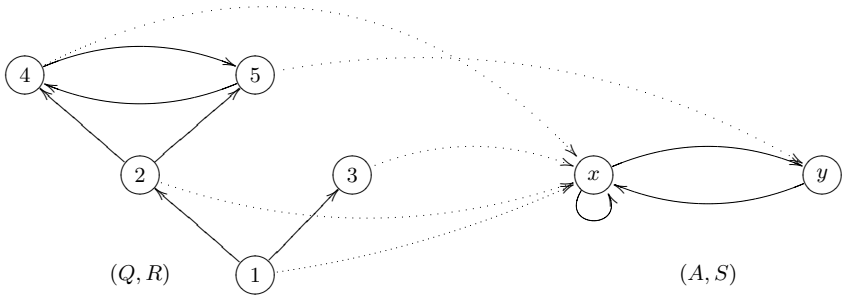


Fig. 1. The concrete and abstract models, and the abstraction relation.

Obviously, we have that $\mathcal{T} \sqsubseteq_\rho \mathcal{A}$. On the other hand, neither ρ-bisimulation nor ρ-completeness hold. In fact, we have that $\mathcal{T} \not\simeq_\rho \mathcal{A}$: $\langle y, 3\rangle \in S^{-1}\rho^{-1}$ whereas $\langle y, 3\rangle \not\in \rho^{-1}R^{-1}$. Also, $\mathcal{T} \not\lesssim_\rho \mathcal{A}$: $\langle 1, y\rangle \in \rho S^{-1}$ whereas $\langle 1, y\rangle \not\in R^{-1}\rho$.

Consider the following temporal specification φ: "at present state there exists an execution trace which visited the state 5", namely, "the present state may be reached from the state 5". Of course, for the concrete model $\mathcal{T}$ we have that $|\varphi|_\mathcal{T} = \{4, 5\}$ and therefore, for example, $2 \not\in |\varphi|_\mathcal{T}$. Let p be the atomic proposition "the present state is 5" so that the corresponding interpretation $\mathcal{I}$ is $\mathcal{I}(p) = \{5\}$. The induced interpretation on $\wp(A)$ is therefore $\mathcal{I}^\sharp \stackrel{\text{def}}{=} \text{post}[\rho] \circ \mathcal{I}$ such that $\mathcal{I}^\sharp(p) = \{y\}$. Then, φ can be specified in the past μ-calculus by the formula $\varphi \stackrel{\text{def}}{=} \mu x.(p \vee \Diamond^p x) \in L_\mu^{\leftarrow}$.

Our aim is to perform an abstract check of $2 \not\in |\varphi|_\mathcal{T}$. Preservation given by Theorem 4.1 is not enough: we need strong preservation and Theorem 4.2 cannot be applied. Thus, we refine the abstract model checking $\mathcal{A}$ using our methodology in order to get an enhanced abstract model which satisfies the hypotheses of Theorem 4.3. Note that in order to be able to apply Theorem 4.3, we will need that the interpretation of the atomic proposition p belongs to $\text{img}(\text{pre}[\rho])$: this is already true for $\mathcal{A}$ and hence it will be still true for the refinement of $\mathcal{A}$.

Let $\mu_\rho \stackrel{\text{def}}{=} \widetilde{\text{pre}}[\rho] \circ \text{post}[\rho]$. Thus, the set of fixpoints of μ_ρ is as follows:

$$\mu_\rho = \{\varnothing, \{1, 2, 3, 4\}, \{5\}, \{1, 2, 3, 4, 5\}\}.$$

It is immediate to observe that μ_ρ is already partitioning, and therefore, by Lemma 6.2, $\mathbb{P}(\mu_\rho) = \mu_\rho$.

By Lemma 6.8, let us now compute $\mathbb{C}_R^s(\mu_\rho)$. Since $\mathbb{C}_R^s(\mu_\rho)$ is an additive closure, it is enough to compute $\mathbb{C}_R^s(\mu_\rho)(\{q\})$ for any singleton $\{q\}$. By Lemma 6.8, let

us recall that $\mathbb{C}^s_R(\mu_\rho)(\{q\}) = \{x \in Q \mid \mu(\mathrm{post}[R^{-1}](\{x\})) = \mu(\mathrm{post}[R^{-1}](\{q\}))\}$. $\mathrm{post}[R^{-1}]$ works as follows:

$$\begin{aligned}
\mathrm{post}[R^{-1}](\{1\}) &= \varnothing,\\
\mathrm{post}[R^{-1}](\{2\}) &= \{1\},\\
\mathrm{post}[R^{-1}](\{3\}) &= \{1\},\\
\mathrm{post}[R^{-1}](\{4\}) &= \{2,5\},\\
\mathrm{post}[R^{-1}](\{5\}) &= \{2,4\}.
\end{aligned}$$

Hence, we have:

$$\begin{aligned}
\mathbb{C}^s_R(\mu_\rho)(\{1\}) &= \{1\},\\
\mathbb{C}^s_R(\mu_\rho)(\{2\}) &= \{2,3,5\},\\
\mathbb{C}^s_R(\mu_\rho)(\{3\}) &= \{2,3,5\},\\
\mathbb{C}^s_R(\mu_\rho)(\{4\}) &= \{4\},\\
\mathbb{C}^s_R(\mu_\rho)(\{5\}) &= \{2,3,5\}.
\end{aligned}$$

and therefore

$$\mathbb{C}^s_R(\mu_\rho) = \{\varnothing, \{1\}, \{4\}, \{1,4\}, \{2,3,5\}, \{1,2,3,5\}, \{2,3,4,5\}, \{1,2,3,4,5\}\}.$$

By Definition 6.6 and Lemma 6.8, we have that

$$\mu_1 = \mu_\rho \sqcap \mathbb{P}(\mu_\rho) \sqcap \mathbb{C}^s_R(\mu_\rho) = \mu_\rho \sqcap \mathbb{C}^s_R(\mu_\rho) =$$

$$= \{\varnothing, \{1\}, \{4\}, \{5\}, \{1,4\}, \{2,3\}, \{1,2,3\}, \{2,3,4\}, \{2,3,5\},$$
$$\{1,2,3,4\}, \{1,2,3,5\}, \{2,3,4,5\}, \{1,2,3,4,5\}\}.$$

Let us now go on with the second iteration, i.e., $\mu_2 = \mu \sqcap \mathbb{P}(\mu_1) \sqcap \mathbb{C}^s_R(\mu_1)$. Observe that μ_1 is not partitioning, because it is not additive: hence,

$$\mathbb{P}(\mu_1) = \{\varnothing, \{1\}, \{4\}, \{5\}, \{1,4\}, \{1,5\}, \{2,3\}, \{4,5\},$$

$$\{1,2,3\}, \{1,4,5\}, \{2,3,4\}, \{2,3,5\},$$

$$\{1,2,3,4\}, \{1,2,3,5\}, \{2,3,4,5\}, \{1,2,3,4,5\}\}.$$

Let us compute $\mathbb{C}^s_R(\mu_1)$. As before, it is enough to compute $\mathbb{C}^s_R(\mu_1)(\{q\})$ for any singleton $\{q\}$. We have:

$$\begin{aligned}
\mathbb{C}^s_R(\mu_1)(\{1\}) &= \{1\},\\
\mathbb{C}^s_R(\mu_1)(\{2\}) = \mathbb{C}^s_R(\mu_1)(\{3\}) &= \{2,3\},\\
\mathbb{C}^s_R(\mu_1)(\{4\}) &= \{4\},\\
\mathbb{C}^s_R(\mu_1)(\{4\}) &= \{5\},
\end{aligned}$$

and therefore, $\mathbb{C}^s_R(\mu_1) = \mu_1$. Thus, $\mu_2 = \mathbb{P}(\mu_1) \sqcap \mathbb{C}^s_R(\mu_1) = \mathbb{P}(\mu_1) \sqcap \mu_1 = \mathbb{P}(\mu_1)$.

It is now easy to check that $\mathbb{P}(\mu_2) = \mu_2$ and $\mathbb{C}^s_R(\mu_2) = \mu_2$. We have reached the fixpoint. Thus, by Corollary 6.7 and Lemma 6.8, μ_2 is the partitioning and complete shell of μ_ρ.

Following Definition 2.6, let us give the abstraction associated to μ_2. We have that $A_{\mu_2} = \{a \stackrel{\text{def}}{=} \mu_2(\{1\}) = \{1\}, b \stackrel{\text{def}}{=} \mu_2(\{2\}) = \mu_2(\{3\}) = \{2,3\}, c \stackrel{\text{def}}{=} \mu_2(\{4\}) = \{4\}, d \stackrel{\text{def}}{=} \mu_2(\{5\}) = \{5\}\}$. The corresponding abstraction $\rho_{\mu_2} \subseteq Q \times A_{\mu_2}$ is depicted in Fig. 2, where the abstract model is equipped with the canonical abstract transition relation $\rho_{\mu_2}^{-1} R \rho_{\mu_2}$. Let $\mathcal{A}_{pc} \stackrel{\text{def}}{=} (A_{\mu_2}, \rho_{\mu_2}^{-1} R \rho_{\mu_2})$.

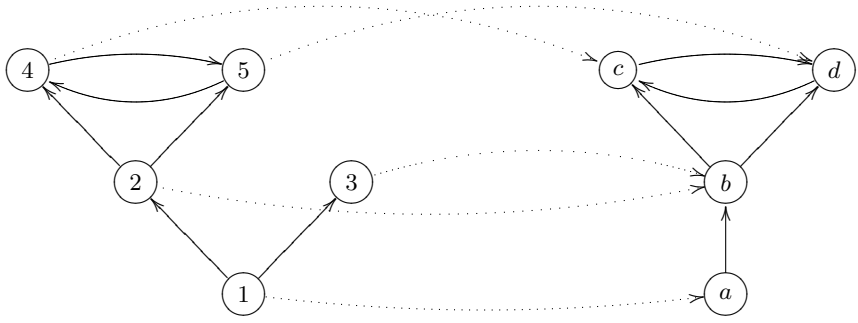


Fig. 2. The refined abstract model $\mathcal{A}_{pc}$.

By Theorem 5.5, $\mathcal{T} \preccurlyeq_{\rho_{\mu_2}} \mathcal{A}_{pc}$. Note that $\text{img}(\text{pre}[\rho_{\mu_2}]) = \{\{1\}, \{2,3\}, \{4\}, \{5\}\}$, and therefore $\mathcal{I}(p) \in \text{img}(\text{pre}[\rho_{\mu_2}])$. The interpretation $\mathcal{I}^\natural$ associated to $\mathcal{A}_{pc}$ is defined as $\text{post}[\rho_{\mu_2}] \circ \mathcal{I}$. Thus, now we can exploit strong preservation as given by Theorem 4.3. We have that $2 \in |\varphi|_{\mathcal{T},\mathcal{I}}$ iff $\text{post}[\rho_{\mu_2}](\{2\}) \subseteq |\varphi|_{\mathcal{A}_{pc},\mathcal{I}^\natural}$. Since $b \not\in |\varphi|_{\mathcal{A}_{pc},\mathcal{I}^\natural}$, by strong preservation we conclude that $2 \not\in |\varphi|_{\mathcal{T},\mathcal{I}}$, as desired.

7 Future Work

As recalled in Section 4, it is well known that bisimulation for the abstract model is a sufficient condition for strong preservation. It is then natural to ask whether it is possible to apply the methodological ideas of this paper for minimally refining an abstract model in order to get bisimulation. Unfortunately, our preliminary results seem to indicate that this cannot be done under reasonable hypotheses. Let us also cite a recent related work by Schmidt [18], who studied some notions of refinement for binary relations and simulations in connection with the strong preservation property. The possible relationship between Schmidt's work and ours is subject for future work.

Acknowledgements. We wish to thank Gilberto Filé who contributed to the early stages of this work. This work has been partially supported by the Italian MIUR Cofin projects "Abstract interpretation, type systems and control-flow analysis" (Cofin2000) and "MEFISTO" (Cofin2001).

References

1. S. Bensalem, Y. Lakhnech, S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In *Proc. CAV'98*, LNCS 1427, pp. 319–331, Springer, 1998.
2. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV'00*, LNCS 1855, pp. 154–169, Springer, 2000.
3. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Progress on the State Explosion Problem in Model Checking. In *Informatics - 10 Years Back, 10 Years Ahead.* LNCS 2000, pp. 176–194, Springer, 2001.
4. E.M. Clarke, O. Grumberg, D.E. Long. Model checking and abstraction. In *Proc. ACM POPL'92*, pp. 342–354, 1992.
5. E.M. Clarke, O. Grumberg and D. Long. Model checking and abstraction. *ACM TOPLAS*, 16(5):1512–1542, 1994.
6. E.M. Clarke, O. Grumberg and D.A. Peled. *Model checking.* The MIT Press, 1999.
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. ACM POPL'77*, pp. 238–252, 1977.
8. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. ACM POPL'79*, pp. 269–282, 1979.
9. P. Cousot and R. Cousot. Temporal abstract interpretation. In *Proc. ACM POPL'00*, pp. 12–25, 2000.
10. D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems. *ACM TOPLAS*, 16(5):1512–1542, 1997.
11. J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Proc. CAV'95*, LNCS 939, pp. 54–69, Springer, 1995.
12. G. Filé, R. Giacobazzi, and F. Ranzato. A unifying view of abstract domain design. *ACM Comput. Surv.*, 28(2):333–336, 1996.
13. R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model checking. In *Proc. SAS'01*, LNCS 2126, pp. 356–373, Springer, 2001.
14. R. Giacobazzi and F. Ranzato. Refining and compressing abstract domains. In *Proc. ICALP'97*, LNCS 1256, pp. 771–781, Springer, 1997.
15. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000.
16. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–36, 1995.
17. F. Ranzato. On the completeness of model checking. In *Proc. ESOP'01*, LNCS 2028, pp. 137–154, Springer, 2001.
18. D.A. Schmidt. Binary relations for abstraction and refinement. In *Proc. Workshop on Refinement and Abstraction*, Japan 1999. To appear in Elsevier Electronic Notes in Computer Science.

Semantics for Abstract Interpretation-Based Static Analyzes of Temporal Properties*

Damien Massé

LIX, École Polytechnique, Palaiseau, France,
masse@lix.polytechnique.fr,
http://www.lix.polytechnique.fr/~masse/

Abstract. Analyzing programs with abstract interpretation often requires to use a combination of forward and backward analyzes. However, the forward analyzes are mostly reachability analyzes, which are often not precise enough when we want to check other temporal properties. In this paper we propose to combine transition systems with temporal properties (with backward modalities) to get "extended" transition systems. We define forward and backward semantics for these systems, and give indications for their combination. These semantics, which use set of trees, can be abstracted to make more precise forward and backward analyzes of programs. The combinations of these analyzes are expected to give better results than the current abstract analyzes.

1 Introduction

Most program analyzes are either "forward" or "backward". "Forward" analyzes start with initial states and abstract traces from these states, simulating possible behaviors of the program. "Backward" analyzes start with final states and simulate the reverse execution of the program. As sets of states (and even more set of traces) are often infinite, and cannot be exactly represented, abstractions are used to represent several states with one *abstract* state. Manipulation on abstract states are done by "abstract functions". *Abstract interpretation* [2,6] is a very general framework to get abstractions and abstract functions and to prove the correctness of an abstract analysis. Fully automatic abstractions to infinite domains, with widening and narrowing [2,5], may be the best solution when the analysis must be without any human interaction.

Analyzes with these abstractions can give imprecise results, sometimes not precise enough to prove the property we want to check (or its negation). In this case, we can inject the result (sound but imprecise) in another analysis, improving the results the analysis would have given. We can do that with a forward and a backward analysis, and repeat the process several times. This combination of analyzes (first described in [2]) gives sound and precise results. However, most of the time, the forward analysis is a reachability analysis. This analysis is

* This work was supported in part by the RTD project IST-1999-20527 DAEDALUS of the European IST FP5 program.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 428–443, 2002.
© Springer-Verlag Berlin Heidelberg 2002

effective when the goal is to detect possible run-time errors (or potential termination). When the goal is to find initial states satisfying more complex temporal properties, however, this reachability analysis is still usable [10], but the results do not seem precise enough to really help the backward analysis. This problem appears because the forward analysis is independent of the temporal property being checked. In this paper we propose to use an extension of transition systems we call "extended transition systems", to combine temporal properties and transition systems. We define a forward semantics for these extended systems, based on sets of trees instead of set of traces, which will be more precise and give a precise concrete semantics for abstract interpretation. We define also a backward semantics, and study combinations of backward and forward analyzes in this new framework.

The first section of this paper will show a small example to explain the motivation of this work. In the second section, we will introduce notations and properties on sets of free trees, which will be used afterward. Then we will define what we call "extended transition systems", and present forward and backward maximal execution semantics for these systems. Then we will show methods to produce an extended transition system from a program (represented as a transition system) and a temporal property. Finally, we will discuss on the combination of forward and backward analyzes by abstraction of the different concrete semantics.

2 Motivations

We will start by the very simple program in imperative language presented in Fig. 1.

(1) Initial state: $I: \ x = y = \text{uninit}$.

```
x:= input in {0,1}
```

(2)

```
y:= random in {0,1}
```

(3)

```
x:= x+y
```

(4) Final states: $F: \ x = 1$

Fig. 1. Simple program

`input` describes a non-deterministic function controllable by the user, `random` a non-deterministic, incontrollable function.

We want to know if, from the initial state (program point (1), `x` and `y` uninitialized), the user can go to final states in F. We can describe this property as $R \neq \emptyset$, with:

$$R \ = \ I \cap \text{lfp}\, \lambda X.(F \cup (\Sigma_3 \cap \widetilde{\text{pre}}(X)) \cup (\Sigma_2 \cap \text{pre}(X)))$$

Σ_i being the set of states at program point (i). R is the set of the initial states from which the user is able to reach a final state.

The combination of backward and forward analyzes, in the framework of abstract interpretation, for this kind of temporal property was described in [10]. If S is the result of the analysis, then $I \cap S \supseteq R$, so that $I \cap S \neq \emptyset$ is a necessary condition for the property $R \neq \emptyset$ to be satisfied. In particular, $I \cap S = \emptyset$ implies that $R = \emptyset$, so that in that case the property is unsatisfiable.

We will use interval analysis [4], which is quite imprecise but very fast (precise relational analyzes like polyhedrons [8] are too expensive for large programs).

The result is given in the table (we start by a forward analysis) described in Fig. 2.

Label	Forward	Combination
(1) x:	non-init.	non-init.
y:	non-init.	non-init.
(2) x:	$[0,1]$	$[0,1]$
y:	non-init.	non-init.
(3) x:	$[0,1]$	$[0,1]$
y:	$[0,1]$	$[0,1]$
(4) x:	$[0,2]$	$[1]$
y:	$[0,1]$	$[0,1]$

Fig. 2. Analysis of the program of Fig. 1. Each line describes the result for a program point and a variable (as the analysis is non-relational). The column "Forward" gives the results for the reachability analysis, which the column "Combination" give the combination of these results with a backward analysis. This column is the fixpoint of the analysis.

We get the combination after the first backward analysis, and the result is too imprecise. We can get more informations in two distinct ways:

- We may give a relation between x and y at point (2), describing the information, given by backward analysis, that we must have $0 \leq x + y \leq 1$. This approach would lead to relational domains. In this particular case, octagons [13] would be sufficient, but more complex cases would need complex relational analyzes. Thus we try another approach.
- It may be better if the information given by the forward analysis enables to give $\emptyset$ already at program point (3) (with the backward analysis). As the command `y := random in {0,1}` in the program does not use `x`, the intuition would be not to modify `x` during the analysis of this command, but during the analysis of `x := y+x`. To get this result, we must know that on the point (2), `y` can take each value in $\{0,1\}$ whatever the value of `x`, and regardless of the choices of the user. On the contrary, the value of `x` depends on the choices of the user. This distinction must appear in the forward semantics.

The forward analysis derives from the trace semantics, which is the most general semantics we can have from a transition relation and a set of initial states. This semantics, which gives set of traces, does not include the context of the execution of the program (here, the difference between `input` and `random`). Our idea is to make several sets of traces, each set describing a "strategy" of the "user" (with the same initial state). Each set of traces forms a tree, so we get a set of trees. With the program used as an example, we get two trees (described in Fig. 4), and none of those has all its leaves satisfying $x = 1$. Thus, there is no way to force $x = 1$ in program point (4).

We can modify the program by swapping the `random` command and the `input` command. The new program is given in Fig. 3. In this case, the choice of the "user" takes place in two possible situations ($x = 0$ or $x = 1$ at program point (2)). As it has two possibilities for this choice, we have four possible strategies. Therefore, we get four trees (see Fig. 4 for detail), each tree expressing a strategy by determining the choices the "user" would make in all situations. One of these trees has all its leaves satisfying $x = 1$, which shows that there is a winning strategy for the "user".

(1) Initial state: $I: \ x = y =$ non-init.

```
x:= random in {0,1}
```

(2)

```
y:= input in {0,1}
```

(3)

```
x:= x+y
```

(4) Final states: $F: \ x = 1$

Fig. 3. Modified program.

This example is a form of game. However, this approach is not specifically related to games. The goal is to prove a temporal property, and each tree may be seen as a potential "proof" of the property. The presence of a winning strategy is merely the proof of the temporal property.

In the following sections, we develop this approach. We must begin by some definitions and notations about trees (or, more accurately, about free trees).

3 Trees

Transition systems traditionally use sets of traces. In order to combine the transition relation with the temporal property being checked, we will use sets of free trees. The set of sets of free trees will be the domain of the semantics of extended transition systems. The following section gives some definitions and notations about this domain.

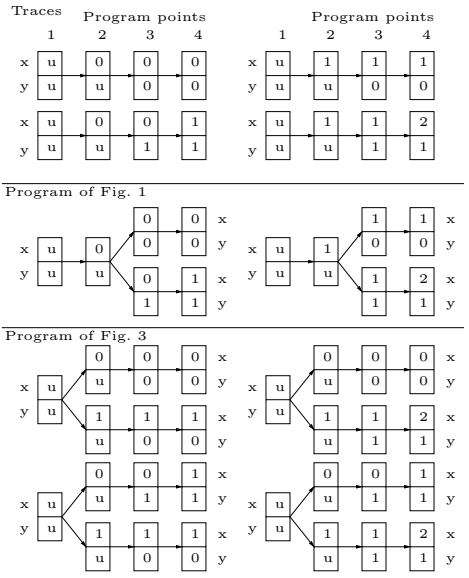


Fig. 4. Traces generated by the programs of Fig. 1 and Fig. 3. Each state is described by its program point (given by its column) and the value of x and y. The traces which describe the same "strategy" for the user are put together in a tree for both program.

3.1 Free Trees

We will note Σ the set of states, Σ^* the set of finite sequences of elements of Σ, Σ^ω the set of infinite sequences, and Σ^∞ the union of Σ^* and Σ^ω. ϵ denotes the empty sequence.

We note $\preceq$ the prefix order on Σ^∞. The dot . is the concatenation operator between two sequences. It is extended to sets of sequences in the following way:

$$\forall u \in \Sigma^*, \forall V \subseteq \Sigma^\infty,\ u.V = \{u.v \mid v \in V\},$$
$$\forall U \subseteq \Sigma^*, \forall V \subseteq \Sigma^\infty,\ U.V = \{u.V \mid u \in U\}.$$

In the following, σ, σ' will always be states, and u, v will always be sequences, possibly empty. For example, $u.\sigma$ will denote a (non-empty) finite sequence that ends with σ.

Definition 1 (Free tree). *A* free tree *labeled by Σ is a non-empty prefix-closed subset of Σ^* with only one "root" (sequence of length one)*[1].

All the trees used in this article will be free trees. As we will never use the empty sequence ϵ of a free tree, we may omit it as well when describing a free tree.

[1] This definition excludes the empty tree $\{\epsilon\}$.

t being a tree, we note $\bar{t}$ the closure of t in Σ^∞, $root(t)$ the "root" of t, and $branch(t)$ the set of maxima of $\bar{t}$. We will name the elements of $branch(t)$ the *branches* of t. A *leaf* of t is the last element of a finite branch of t. The set of leaves of t will be noted $leaf(t)$.

We denote by $\mathcal{T}$ the set of free trees labeled by Σ, and $\Theta = \wp(\mathcal{T})$.

Definition 2 (Subtrees). *Let t be a tree, and $u.\sigma \in t$, we define the* subtree *of t rooted at $u.\sigma$ as:*

$$t_{[u.\sigma]} = \{\sigma.v \mid u.\sigma.v \in t\}.$$

Then $t_{[u.\sigma]}$ is a tree rooted by σ. The set of subtrees of a tree t is denoted $subtrees(t)$.

Definition 3 (Well-founded tree). *A free tree t labeled by Σ is said to be well-founded iff $branch(t) \subseteq \Sigma^*$.*

A well-founded tree does not have infinitely increasing chains of sequences.

A tree with finite arity is well-founded if and only if it is finite. However, in this article we will deal mostly with infinite arity, and the distinction between finite and well-founded must be made (an infinite but well-founded tree is presented in Fig. 5). We will note $\mathcal{T}_{WF}$ the set of well-founded trees labeled by Σ.

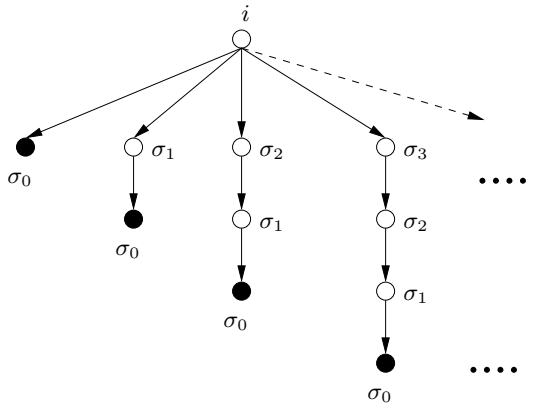


Fig. 5. An infinite well-founded tree. $\Sigma = \{i, \sigma_n \mid n \geq 0\}$, and $t = \{i\sigma_k\sigma_{k-1}\ldots\sigma_l \mid 0 \leq l \leq k\}$ The depth of the tree is unbounded, but it has no infinite branch.

In order to define some sets of trees by conditions on their infinite branches, or on their well-founded subtrees, we introduce some notations:

Definition 4. – *With $\theta \subseteq \mathcal{T}_{WF}$, we note $\mathcal{T}_\theta$ the set of trees t such that all well-founded subtrees of t are in θ:*

$$\mathcal{T}_\theta = \{t \in \mathcal{T} \mid subtrees(t) \cap \mathcal{T}_{WF} \subseteq \theta\};$$

– *With $S \subseteq \Sigma^\omega$, we note $\mathcal{T}^S$ the set of trees t such that each infinite branch of t has a suffix in S:*

$$\mathcal{T}^S = \{t \in \mathcal{T} \mid \forall u \in \mathit{branch}(t) \cap \Sigma^\omega, \exists v \in S, u = u'.v\}.$$

3.2 Orders on Set of Trees

$\subseteq$ is an partial order on $\mathcal{T}$. However, this order is too imprecise: we will define a "prefix" order $\preceq_{\mathcal{T}}$ on trees (as in [9]), such that $t \preceq_{\mathcal{T}} t'$ if t' extends t only on leaves of t:

$$t \preceq_{\mathcal{T}} t' \iff t \subseteq t' \wedge (\forall u' \in \mathit{branch}(t'), \exists u \in \mathit{branch}(t), u \preceq u').$$

We denote by $\sqsubseteq$ the preorder on Θ defined as:

$$\theta \sqsubseteq \theta' \iff (\forall\, t' \in \theta',\ \exists\, t \in \theta,\ t \preceq_{\mathcal{T}} t')$$

Note that $\emptyset$ is the only supremum in $(\Theta, \sqsubseteq)$ (and, in general, $\theta \subseteq \theta' \Rightarrow \theta' \sqsubseteq \theta$).

As $\sqsubseteq$ is a preorder, it is not usable to define semantics as fixpoints of iterative sequences. Rather than quotienting Θ, we will restrict ourselves to a subset of Θ in which $\sqsubseteq$ will be a partial order. The subset used will be the sets of non-comparable trees where all comparable branches of each tree are equal: the branches of the trees are traces, and all traces must be equally complete in all the trees where they appear. With this restriction (stronger than the quotienting), we can define a least upper bound for increasing sequences of sets.

Proposition 1. *We note:*

$$\Theta_{NC} = \{\theta \in \Theta \mid \forall (t,t') \in \theta^2, \forall u \in \mathit{branch}(t), \forall u' \in \mathit{branch}(t'), u \preceq u' \Rightarrow u = u'\}.$$

And, $(\theta_i)_{i\in\mathbb{N}}$ being an increasing chain of Θ_{NC}, we define $\sqcup(\theta_i)$ as:

$$\in \sqcup(\theta_i) \iff \exists (t_0, t_1, \dots) \in \theta_0 \times \theta_1 \times \dots,$$
$$t_0 \preceq_{\mathcal{T}} t_1 \preceq_{\mathcal{T}} \dots \wedge t = \cup(t_i).$$

Then $\Theta_{NC}\langle \sqsubseteq, \bot, \sqcup\rangle$ is a complete partial order *(cpo) with $\bot = \{\{\sigma\} \mid \sigma \in \Sigma\}$.*

We will therefore be able to define semantics as limit of increasing chains for $\sqsubseteq$ in Θ_{NC}. In the next section, we define the extension of the transition systems we want to study, and forward and backward semantics for this extension.

4 Extended Transition System

4.1 Definitions

As a transition relation associates a set of successors to each state, an extended transition relation will associate several sets of successors to each state.

Definition 5 (Extended transition relation and system). *Σ being a set of states, an extended transition relation τ on Σ is a subset of $\Sigma \times \wp(\Sigma)$, or an element of $\Sigma \rightarrow \wp(\wp(\Sigma))$. An extended transition system is a pair $\langle \Sigma, \tau \rangle$, where τ is an extended transition relation on Σ.*

We will use the functional form for extended transition relations. We can interpret an extended transition system in two ways: first as a program with two players (the user and the machine), such that from a state σ, the user choose the set of potential next states in $\tau(\sigma)$, and the machine arbitrarily takes one state in the chosen set. The second approaches is related to logic: each set in $\tau(\sigma)$ is an alternative way (expressed as a set of requirements) to prove σ. In this approach, $\tau(\sigma) = \{\emptyset\}$ means that σ is an axiom, whereas $\tau(\sigma) = \emptyset$ means that σ is false.

Each extended transition relation can be described as a set of "elementary" trees of depth 1 or 2, in the following way:

Definition 6 (Elementary trees). *τ being an extended transition relation, we denote by* $\mathrm{elem}(\tau)$ *the set of trees:*

$$\mathrm{elem}(\tau) = \{\{\sigma,\ \sigma.\sigma' \mid \sigma' \in S\} \ \mid \sigma \in \Sigma,\ S \in \tau(\sigma)\}$$

$\mathrm{elem}(\tau)$ gives a graphical description of the extended transition relation τ (see Fig. 6 for example).

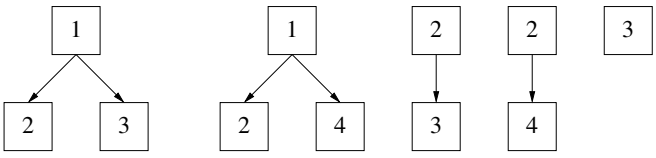


Fig. 6. The elementary trees of the extended transition system $\langle \Sigma, \tau \rangle$ with $\Sigma = \{1, 2, 3, 4\}$ and $\tau(1) = \{\{2, 3\}, \{2, 4\}\}$, $\tau(2) = \{\{3\}, \{4\}\}$, $\tau(3) = \{\emptyset\}$, $\tau(4) = \emptyset$.

4.2 Forward Semantics

From a set of trees, we make a forward step by appending elementary trees to the leaves of the trees:

Definition 7 (Forward operator). *The forward operator F of an extended transition system $\langle \Sigma, \tau \rangle$ is:*

$$\begin{aligned} F\ :\ \Theta \ &\rightarrow\ \Theta \\ \theta \ &\mapsto \{t' \ \mid \exists t \in \theta, \\ &\qquad \exists s_{u.\sigma} \in \tau(\sigma) \textit{ for all } u.\sigma \in \textit{branch}(t) \textit{ s.t.} \\ &\qquad t' = t \cup \{u.\sigma.\sigma' \mid u.\sigma \in \textit{branch}(t), \sigma' \in s_{u.\sigma}\}\}. \end{aligned}$$

$F(\theta)$ is created by appending (coherently) an elementary tree to each finite branch of each tree of θ. We can see that F can also "remove" a tree when a leaf σ satisfies $\tau(\sigma) = \emptyset$. F is, of course, a morphism for the union $\cup$. A simple forward semantics of (Σ, τ) is therefore $\mathrm{lfp}\, \lambda X.(\mathcal{I} \cup F(X))$, where $\mathcal{I} = \{\{i\} \mid i \in I\}$ is the set of initial states described as trees:

Definition 8 (Partial forward semantics). *The partial forward semantics $\mathcal{F}^p(I)$ of an extended transition system $\langle \Sigma, \tau \rangle$ with initial states $I \subseteq \Sigma$ is defined as:*

$$\mathcal{F}^p(I) = \mathrm{lfp}_{\emptyset}^{\subseteq}\, \lambda X.(\mathcal{I} \cup F(X))$$

where $\mathcal{I} = \{\{i\} \mid i \in I\}$.

With traces, this semantics gives the partial traces of the transition system starting from the initial states.

A more expressive semantics would give only the maximal trees coherent with the transition system. We can achieve this goal by using the order $\sqsubseteq$ on sets of incomparable trees. We need a condition on τ:

Proposition 2. *If an extended transition system $\langle \Sigma, \tau \rangle$ satisfies:*

$$\forall \sigma \in \Sigma : \ \emptyset \in \tau(\Sigma) \implies \tau(\sigma) = \{\emptyset\}, \tag{1}$$

then F defines a $\sqsubseteq$-monotonic and extensive function from Θ_{NC} to Θ_{NC}.

This condition says that a potential final state is always final. Note that a tree which has a leaf σ with $\tau(\sigma) = \emptyset$ will disappear in the next iteration of F. This is compatible with the order $\sqsubseteq$, which has $\emptyset$ as a supremum.

Under these conditions, we can define a maximal forward semantics on an extended transition system ($\mathrm{lfp}_a^{\sqsubseteq} F$ is the $\sqsubseteq$-fixpoint of F greater than or equal to a):

Definition 9 (Maximal forward semantics). *The maximal forward semantics $\mathcal{F}^m(I)$ of an extended transition system $\langle \Sigma, \tau \rangle$ with initial states $I \subseteq \Sigma$ is defined as:*

$$\mathcal{F}^m(I) = \mathrm{lfp}_{\mathcal{I}}^{\sqsubseteq} F$$

with $\mathcal{I} = \{\{i\} \mid i \in I\}$.

An example of the maximal forward semantics is given in Fig. 7.

4.3 Backward Semantics

There are two possible approaches for the backward semantics: a least fixpoint, which gives only well-founded trees, or a greatest fixpoint, which gives all maximal trees. Both will be described here. The possibility of using both with a bi-inductive definition, as we can do with traces, will be discussed afterward.

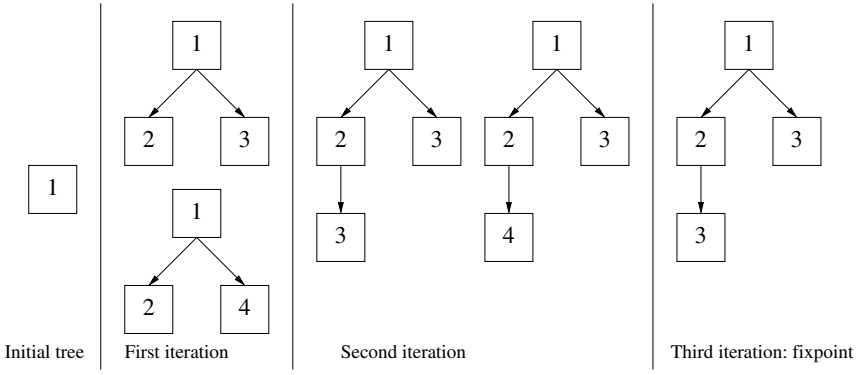


Fig. 7. The iterations and the maximal forward semantics of the extended transition system given in Fig. 6, with $I = \{1\}$.

Definition 10 (Backward operator). *The backward operator B of an extended transition system $\langle \Sigma, \tau \rangle$ is:*

$$\begin{aligned} B : \Theta &\to \Theta \\ \theta &\mapsto \{ \in \theta \mid \sigma = root(t) \\ &\quad \exists S \in \tau(\sigma), \exists t_s \in \theta \text{ for all } s \in S \text{ with } root(t_s) = s, \\ &\quad t = \sigma.\{t_s\}\}. \end{aligned}$$

$B(\theta)$ are the trees created by appending the trees of θ to an elementary tree of τ. B is monotonic and a complete $\cap$-morphism. However, B is not continuous.

To define the maximal trace semantics (as in [7]), we start from all final states (states which have no successor). Here we will start with states σ such that $\tau(\sigma) = \{\emptyset\}$ (on the contrary, states σ such that $\tau(\sigma) = \emptyset$ are more like error states). Thus we note:

$$f_\tau = \{\sigma \in \Sigma \mid \tau(\sigma) = \{\emptyset\}\}$$

With traces, we define the maximal finite traces semantics (with a least fixpoint), and a maximal traces semantics (finite or infinite) with a greatest fixpoint. Here we define the maximal well-founded backward semantics, and the maximal backward semantics.

Definition 11 (Maximal well-founded backward semantics). *The maximal well-founded backward semantics is defined as:*

$$\mathcal{B}^m_{WF} = \mathrm{lfp}^{\subseteq} \lambda X.(f_\tau \cup B(X)).$$

Note that, as B is not continuous, the least fixpoint may be not reached after ω iterations. This may appear in the case of unbounded non-determinism.

The well-founded backward semantics can be defined as a greatest fixpoint, starting only from the well-founded trees. In this case, the fixpoint is reached in ω iterations (as most).

Proposition 3.

$$\mathcal{B}^m_{WF} = \mathrm{gfp}^{\subseteq}_{\mathcal{T}_{WF}} \lambda X.(f_\tau \cup B(X)).$$

Definition 12 (Maximal backward semantics). *The maximal backward semantics is defined as:*

$$\mathcal{B}^m = \mathrm{gfp}^{\subseteq} \lambda X.(f_\tau \cup B(X)).$$

The following theorem displays the links between backward and forward semantics.

Theorem 1. *We have, for all $I \subseteq \Sigma$, with $\mathcal{T}(I)$ being the set of trees rooted by an element of I:*

$$\mathcal{B}^m_{WF} \cap \mathcal{T}(I) = \mathcal{F}^m(I) \cap \mathcal{T}_{WF};$$

$$\mathcal{B}^m \cap \mathcal{T}(I) = \mathcal{F}^m(I).$$

Cousot [3,7] define the maximal traces semantics of a transition system as a mix of a greatest and a least fixpoint (the greatest fixpoint being used for infinite traces, and the least fixpoint being used for finite traces). This combination is useful for further abstractions such as potential termination [3]. However, in the case of sets of trees, infinite and finite traces are mixed, and it seems quite hard to define the maximal backward semantics as a combination of a greatest and a least fixpoint. However, we can compute first only well-founded trees with a least fixpoint, and then use a greatest fixpoint on a restricted set of infinite trees defined by the well-founded set previously computed:

Proposition 4. *We have:*

$$\mathcal{B}^m = \mathrm{gfp}^{\subseteq}_{\mathcal{T}_{\mathcal{B}^m_{WF}}} \lambda X.(f_\tau \cup B(X)).$$

(we recall that $\mathcal{T}_{\mathcal{B}^m_{WF}}$ is the set of trees which have all subtrees in $\mathcal{B}^m_{WF}$).

Thus, to compute the maximal backward semantics, we first generate the well-founded backward semantics as a least fixpoint, then create the set of all trees with well-founded subtrees in the generated set, and then use a greatest fixpoint.

5 Making Extended Transition System

Our idea is to create an extended transition system from a program and a temporal property we want to prove. A program is represented by a classical transition system (Σ_0, τ_0). The transformation depends on the temporal property. In this section, we present some examples.

5.1 Distinction between Two Non-determinisms

The temporal properties expressed here are quite easy, with only one fixpoint. Using the μ-calculus formalism, the temporal property is written

$$\Phi = \alpha X.(A \vee B \wedge \Diamond X \vee C \wedge \Box X)$$

with α being either μ or ν.

This formula includes, of course, all the basic CTL operators (excepted **AX** and **EX**). But we do not intend to combine them (with several fixpoints). However, the formulas express also some form of game properties, by introducing "fated" and "free" non-determinisms, depending of the current state.

States satisfying A are "final" states. States satisfying B are "free" non-deterministic states: we can choose the successor in order to get A (in other words, if one successor satisfy Φ, then the state satisfy Φ). States satisfying C are "fated" non-deterministic states: we cannot choose the successor. States which satisfy neither A, B nor C are errors states. For the sake of simplicity, we identify A with the set of states satisfying A.

The extended transition system created $\langle \Sigma, \tau \rangle$ is defined as follows, with $\Sigma = \Sigma_0$:

$$\begin{array}{ll} \tau(\sigma) = \emptyset & \text{if } \sigma \notin A \cup B \cup C \\ \tau(\sigma) = \{\emptyset\} & \text{if } \sigma \in A \\ \tau(\sigma) = \{\{\sigma'\} \mid \sigma' \in \tau_0(\sigma)\} & \text{if } \sigma \in B \\ \tau(\sigma) = \{\tau_0(\sigma)\} & \text{if } \sigma \in C \end{array}$$

τ satisfies (1). Furthermore, the roots of the backward semantics with a least fixpoint (resp. with a greatest fixpoint) of $\langle \Sigma, \tau \rangle$ are exactly the set of states satisfying Φ with $\alpha = \mu$ (resp. with $\alpha = \nu$).

Theorem 2. *An initial state satisfies Φ (with $\alpha = \mu$) if and only it is the root of a well-founded tree in $\mathcal{F}^m(I)$.*

An initial state satisfy Φ (with $\alpha = \nu$) if and only it is the root of a tree in $\mathcal{F}^m(I)$.

5.2 CTL

A CTL formula has several nested fixpoints [1]. We may simulate the results of these fixpoints with only one fixpoint by extending the set of states. Let us recall the basic CTL operators (the negation is only used for atomic predicates):

$$\begin{aligned} \phi = p \mid \neg p \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \mathbf{AX}\phi \mid \mathbf{EX}\phi \mid \mathbf{AF}\phi \mid \mathbf{EF}\phi \\ \mid \mathbf{AG}\phi \mid \mathbf{EG}\phi \mid \mathbf{A}(\phi_1 \mathbf{U} \phi_2) \mid \mathbf{E}(\phi_1 \mathbf{U} \phi_2) \mid \mathbf{A}(\phi_1 \mathbf{R} \phi_2) \mid \mathbf{E}(\phi_1 \mathbf{R} \phi_2). \end{aligned}$$

Φ being a formula, we denote $sub(\Phi)$ the set of all sub-formulas in Φ. Then we define Σ as:

$$\Sigma = \Sigma_0 \times sub(\Phi).$$

The extended transition relation τ is defined in Fig. 8. Let Ψ be the set of subformulas of Φ of the form **AG**, **EG**, **AR**, **ER**. For all ϕ in Ψ, we define $T_\phi = (\Sigma_0 \times \{\phi\})^\omega$, and $T_\Psi = \bigcup_{\phi \in \Psi} T_\phi$. T_Ψ are the infinite sequences where the temporal formula is constant and of the form **G** or **R**. The only infinite branches allowed for a tree "proving" Φ must have a suffix in T_Ψ.

Then an initial state i satisfy Φ if and only if (i, Φ) is the root of a tree in $\mathcal{F}^m(I \times \Phi) \cap \mathcal{T}^{T_\Psi}$.

$$
\begin{aligned}
\tau(\sigma, p) &= \{\emptyset\} \text{ if } \sigma \text{ satisfy } p \\
\tau(\sigma, p) &= \emptyset \text{ otherwise} \\
\tau(\sigma, \phi_1 \vee \phi_2) &= \{\{(\sigma, \phi_1)\}, \{(\sigma, \phi_2)\}\} \\
\tau(\sigma, \phi_1 \wedge \phi_2) &= \{\{(\sigma, \phi_1), (\sigma, \phi_2)\}\} \\
\tau(\sigma, \mathbf{AX}\phi) &= \{\{(\sigma', \phi) \mid \sigma' \in \tau_0(\sigma)\}\} \\
\tau(\sigma, \mathbf{EX}\phi) &= \{\{(\sigma', \phi)\} \mid \sigma' \in \tau_0(\sigma)\} \\
\tau(\sigma, \mathbf{AF}\phi) &= \{\{(\sigma, \phi)\}, \{(\sigma', \mathbf{AF}\phi) \mid \sigma' \in \tau_0(\sigma)\}\} \\
\tau(\sigma, \mathbf{EF}\phi) &= \{\{(\sigma, \phi)\}, \{(\sigma', \mathbf{EF}\phi)\} \mid \sigma' \in \tau_0(\sigma)\} \\
\tau(\sigma, \mathbf{AG}\phi) &= \{\{(\sigma, \phi), (\sigma', \mathbf{AF}\phi) \mid \sigma' \in \tau_0(\sigma)\}\} \\
\tau(\sigma, \mathbf{EG}\phi) &= \{\{(\sigma, \phi), (\sigma', \mathbf{AF}\phi)\} \mid \sigma' \in \tau_0(\sigma)\} \\
\tau(\sigma, \mathbf{A}(\phi_1 \mathbf{U} \phi_2)) &= \{\{(\sigma, \phi_2)\}, \{(\sigma, \phi_1), (\sigma', \mathbf{A}(\phi_1 \mathbf{U} \phi_2)) \mid \sigma' \in \tau_0(\sigma)\}\} \\
\tau(\sigma, \mathbf{E}(\phi_1 \mathbf{U} \phi_2)) &= \{\{(\sigma, \phi_2)\}, \{(\sigma, \phi_1), (\sigma', \mathbf{E}(\phi_1 \mathbf{U} \phi_2))\} \mid \sigma' \in \tau_0(\sigma)\} \\
\tau(\sigma, \mathbf{A}(\phi_1 \mathbf{R} \phi_2)) &= \{\{(\sigma, \phi_1), (\sigma, \phi_2)\}, \{(\sigma, \phi_2), (\sigma', \mathbf{A}(\phi_1 \mathbf{R} \phi_2)) \mid \sigma' \in \tau_0(\sigma)\}\} \\
\tau(\sigma, \mathbf{E}(\phi_1 \mathbf{R} \phi_2)) &= \{\{(\sigma, \phi_1), (\sigma, \phi_2)\}, \{(\sigma, \phi_2), (\sigma', \mathbf{E}(\phi_1 \mathbf{R} \phi_2))\} \mid \sigma' \in \tau_0(\sigma)\}
\end{aligned}
$$

Fig. 8. Definition of τ for CTL.

6 Combination

One of the basic goal of using extended transition systems is to make automatic abstractions, and to use the combination between forward and backward analyzes. The principle of the combination is to use the result of the previous analysis at each iteration of the new analysis, to get more precise values (which are still sound). Even if the operation is done on the abstract domain, there are underlying operations on the concrete domain. In this section, we examine the possible combinations one can see in the concrete domain. As the approximations are over-approximations[2], the operations must be made so that we still get

[2] The combination of analyzes uses over-approximations to check *intersection* of properties (e.g. states which are initial states *and* satisfy the temporal property). To get an equivalent of a lower approximation, we can use the negation of the temporal property.

a superset of $\mathcal{F}^m(I)$ (or $\mathcal{F}^m(I) \cap \mathcal{T}_{WF}$) at the end of the computation, in order to keep sound results.

6.1 Using Backward Results in Forward Analysis

For each iteration of F, we can remove the trees which are not a subset of a tree obtained in the backward analysis:

Theorem 3. *With* $\downarrow : \Theta \times \Theta \longrightarrow \Theta$ *defined as:*

$$\theta_1 \downarrow \theta_2 = \{t_1 \in \theta_1 \mid \exists t_2 \in \theta_2, t_1 \subseteq t_2\}$$

we have the following results:

$$\mathcal{F}^m(I) = \mathrm{lfp}_{\mathcal{I}}^{\sqsubseteq} \lambda X.(F(X) \downarrow \mathcal{B}^m)$$
$$\mathcal{F}^m(I) \cap \mathcal{T}_{WF} = \mathrm{lfp}_{\mathcal{I}}^{\sqsubseteq} \lambda X.(F(X) \downarrow \mathcal{B}^m_{WF})$$

6.2 Using Forward Results in Backward Analysis

Each iteration of the backward analysis gives a set of trees. One possible improvement is to intersect this set with a set of potential trees given by the forward analysis. However, it would be much better to impose some kind of *constraints* on the set of trees given by the iteration. The idea is to remove trees which will not change the final results because they don't appear together with other tree. Formally, from $\mathcal{F}^m(I)$, we would like to get $\mathcal{H} \subseteq \Theta$ such that B' defined as:

$$B'(\theta) = \{t \in B(\theta) \mid \exists H \in \mathcal{H}, t \in H \wedge H \subseteq B(\theta)\}$$

may replace B as the backward operator.

In order to get $\mathcal{H}$, we need to distinguish the different cases of backward semantics:

With a greatest fixpoint. To get t in $\mathcal{B}^m$, all sub-trees of t must be in $\mathcal{B}^m$, at each stage of the iteration. Therefore, we can take for H the sets of all sub-trees of each tree in $\mathcal{F}^m(I)$:

Theorem 4. *With* $\mathcal{H} = \{subtrees(t) \mid t \in \mathcal{F}^m(I)\}$, *we have:*

$$\mathcal{B}^m = \mathrm{gfp}^{\subseteq} \lambda X.(f_\tau \cup B'(X)).$$

With a least fixpoint. Let $\mathcal{B}$ be an iteration of $\lambda X.(f_\tau \cup B(X))$. Then t will be in $\mathcal{B}^m_{WF}$ if we can find a set θ of subtrees of t in $\mathcal{B}$ which "covers" all the branches of t:

$$\forall u \in t, \exists v \in t,\ u \preceq v \vee v \preceq u, \text{ such that } t_{[v]} \in \theta \tag{2}$$

To see what may be θ graphically, we just remove an "upper part" of t without removing a complete branch. We get a set of trees which satisfy the property (2) (see Fig. 9 for an example). This result is expressed in the following theorem:

Theorem 5. *We define a "slice" θ of a tree t as a set of subtrees of t which satisfies (2). We denote Slices(t) the set of all slices of a tree t. Then, with*

$$\mathcal{H} = \bigcup_{t \in (\mathcal{F}^m(I) \cap \mathcal{T}_{WF})} Slices(t)$$

we have:

$$\mathcal{B}^m_{WF} = \mathrm{lfp}^{\subseteq} \lambda X.(f_\tau \cup B'(X))$$

In practice, this theorem is not usable, $\mathcal{H}$ is too large, and its approximations would be too general. Finding a smaller suitable set is a part of future developments.

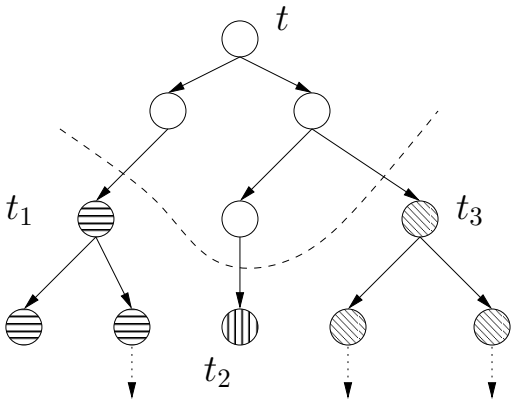


Fig. 9. An example of a set of trees ($\{t_1, t_2, t_3\}$) which satisfies (2) with respect to the tree t.

7 Conclusion

We have defined new forward and backward concrete semantics for checking temporal properties on programs: from the transition system and the temporal property, we create an extended transition system which combines informations from both, and we give the semantics of this extended transition system. The forward semantics, more than the backward semantics, use a complex order on set of trees, and complex operations. Deriving analyzes from these semantics may be hard. However, our goal is not to prove properties directly from the forward analyzes, but to collect more informations which will help the backward analyzes.

Much work about approximations of sets of trees for abstract interpretation was done by Mauborgne [11,12]. Even if this work deals with trees with finite arity, it is expected to be very helpful to abstract structures on sets of trees.

Acknowledgments. I wish to thank the anonymous referees, as well as Radhia and Patrick Cousot, Francesco Logozzo, Charles Hymans and Jean Goubault-Larrecq for their comments and suggestions.

References

1. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT press, 1999.
2. P. Cousot. *Méthodes itératives de construction et d'approximation de point fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes.* Thèse ès sciences mathématiques, University of Grenoble, March 1978.
3. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.
4. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.
5. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
6. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.
7. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the Ninthteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–94, Albuquerque, New Mexico, January 1992. ACM Press, New York, NY.
8. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.
9. Panagiotis Manolios and Richard J. Trefler. Safety and liveness in branching time. In *Logic in Computer Science*, pages 366–, 2001.
10. D. Massé. Combining backward and forward analyses of temporal properties. In O. Danvy and A. Filinski, editors, *Proceedings of the Second Symposium PADO'2001, Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Sciences*, pages 155–172, Århus, Denmark, 21 – 23 May 2001. Springer-Verlag, Berlin, Germany.
11. Laurent Mauborgne. *Representation of Sets of Trees for Abstract Interpretation.* PhD thesis, École Polytechnique, 1999.
12. Laurent Mauborgne. An incremental unique representation for regular trees. *Nordic Journal of Computing*, 7(4):290–311, 2000.
13. A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001. `http://www.di.ens.fr/~{}mine/publi/article-mine-padoII.pdf`.

Checking Safety Properties of Behavioral VHDL Descriptions by Abstract Interpretation*

Charles Hymans

LIX, École Polytechnique, 91128 Palaiseau, France
chymans@lix.polytechnique.fr

Abstract. We are interested in the automatic verification of digital designs specified in the popular hardware description language VHDL. This paper presents a static analysis that computes a superset of the states maybe reached during the simulation of a VHDL design. We follow the methodology of abstract interpretation. To model the execution of a VHDL description, we first define a concise structural operational semantics. Our analysis is then derived by abstraction from this formal model. It is designed so as to be parametric in the representation of sets of states. Hence, trade-offs between cost and precision can be made by plugging in different abstract domains. This is of particular importance in the case of hardware verification, where one of the major obstacle to the integration of automatic tools in the design flow is the state-explosion problem they face. We instantiate our analysis with a domain that consists in a collection of vectors of constants and whose size is linear in the size of the unit under verification. Among other things, our analysis allows us to assert safety properties.

1 Introduction

During the past few years, hardware designs have seen a dramatic increase in complexity. In consequence, the validation phase now creates a substantial bottleneck in the development cycle. Tools that can reduce the cost of design verification and still be seamlessly integrated in the design flow have thus become of great value.

In this respect, model-checking techniques in its various forms (for instance [9,7,3,24]) have been very successful. We wish to improve upon these techniques by addressing some of the issues they face. First, model-checkers do not directly operate on the system description, but rather on a simplified, finite model. Current hardware description languages such as VHDL [1], Verilog [22] and SystemC [21] all have very intricate semantics. It is, hence, difficult to extract a model from the source code. This step, usually performed manually, creates a semantical gap between the executed code and the verified model. Automatic extraction methods are not able to perform as aggressive abstractions as verification experts. They can lead to very large models that model-checking procedures may

* This work was supported in part by the RTD project IST-1999-20527 DAEDALUS of the european IST FP5 programme.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 444–460, 2002.
© Springer-Verlag Berlin Heidelberg 2002

not always be able to handle. This brings us to our second point, namely, the infamous state-explosion problem. In fact, the design state space is not only huge but also sometimes infinite. Currently, verification efforts are focused on low level descriptions, where the design's structure is well identified: it is an interconnection of a finite number of boolean gates and memory elements. However, before a design reaches this point, the system's behavior is described at a more abstract level. At this level, infinite data-types, such as integers, are commonly used. At last, model-checkers require the verification team to master other formalisms than the design language in order to specify the properties to check.

Our goal is to devise a static analysis for cycle-accurate behavioral VHDL descriptions. The simulation of such descriptions amounts to the parallel execution of a fixed number of sequential processes. Communication between these processes occurs through global variables (the signals) at synchronization points only. Synchronization happens when all processes are suspended (on a wait statement). Signals may be integer or boolean. This notion of execution is formalized as a structural operational semantics. Our analysis computes a superset of the states that may be reached during simulation of the description. Our algorithm is derived by abstraction from the operational simulation semantics and shown to be sound. We therefore bridge the gap between the verification procedure and the source code. The analysis is parametric in the underlying representation of set of states. Different abstract domains may be plugged in and allow to tune the cost/precision ratio. The ability to do very blunt but conservative approximations gives us an opportunity to handle prohibitively large or even infinite state space. However, our algorithm sometimes gives false negative answers[1]. We present a domain whose size is linear in the number of variables in the verified system (but exponentially large in the size of the property). We hope it achieves a good balance between complexity and the amount of false negative answers. Finally, the properties to check need not be written in a distinct formalism. They are expressed in VHDL as monitors and can be used in a more traditional simulation context if the analysis fails. We are limited to safety properties.

Our contribution essentially consists in a coherent integration of disparate techniques. Our concrete semantics for VHDL is greatly inspired by [14]. For the abstract domain, we refined a data-structure used in the constant-propagation algorithm [18] thanks to partitioning [11]. The idea for this representation is rooted in the abstraction of symbolic trajectory evaluation [8]. Also, the essential theoretical framework required to glue these results together is abstract interpretation [10,11].

Organization of the paper. Section 2 introduces the simulation model of behavioral VHDL. We reveal the inherently sequential nature of VHDL in Sect. 3. This leads to the construction of an optimized version of the semantics. The parametric static analysis is described in Sect. 4. It is instantiated in the last section with a particular abstract domain. We then draw our conclusions and discuss future work.

[1] also called false alarms.

2 Simulation Semantics of VHDL

The VHSIC[2] Hardware Description Language, VHDL for short, is one of the most widely used hardware description language. It is a very large language whose different subsets tackle various aspects of system modeling. We consider cycle-accurate behavioral VHDL descriptions. This approximately corresponds to the behavioral synthesis subset which is currently being standardized [2]. For the sake of simplicity and brevity of this exposition, we avoid resolution functions and local variables. These features can be easily handled. However, we impose one essential restriction, that would require some major changes in the semantics to lift: we forbid the use of after clauses in signal assignment statements. We furthermore suppose that the elaboration, as defined in Sect. 12 of the VHDL Language Reference Manual, has been performed. This procedure flattens the hierarchical structure of the design.

This allows us to study the kernel language of Fig. 1. A program consists in the parallel composition of a fixed number of sequential processes. Processes' code is made out of the concatenation of the infinite loop, signal assignment, wait and if statements. The meaning of a VHDL description is given in English prose by the standard [1] as a simulation algorithm. We formalize it by a small-step operational semantics [23]. Formalizations close to ours can be found in [14] and [13].

$$
\begin{array}{rcll}
Prog & ::= & P_1 \mid \ldots \mid P_n & \text{(Parallel composition)} \\
P & ::= & C; P & \text{(Command)} \\
 & \mid & \epsilon & \text{(End of a process)} \\
C & ::= & \texttt{proc} & \text{(Loop)} \\
 & \mid & \texttt{x <= } e & \text{(Signal assignment)} \\
 & \mid & \texttt{wait on } W \texttt{ for } t & \text{(Suspension)} \\
 & \mid & \texttt{if } b \texttt{ then } P_1 \texttt{ else } P_2 & \text{(Selection)} \\
b, e & ::= & n \mid \texttt{true} \mid \texttt{false} \mid \texttt{x} \mid \texttt{random} \mid e_1 \texttt{ + } e_2 & \\
 & \mid & e_1 \texttt{ > } e_2 \mid \texttt{not } b \mid b_1 \texttt{ and } b_2 \mid b_1 \texttt{ or } b_2 &
\end{array}
$$

where `x` is a signal; W is a possibly empty set of signals; t is a positive integer or ∞ (to denote the absence of timeout clause); n is an integer.

Fig. 1. Syntax.

A state of execution is a tuple $(p_1, \ldots, p_n, \rho)$, where each p_i is the local state of the ith process, and ρ is the global environment. We call S the set of all the signals in the program. The global environment ρ maps every signal in S to its current value. All signals have a type, which is either boolean or integer. Naturally, signals can only take values of their type. The local state $p_i = (c_i, \delta_i)$ of the ith process can be further broken down into two parts. The control point c_i of a running process is simply the piece of code P_i that is left to be executed. In

[2] VHSIC stands for Very High Scale/Speed Integrated Circuits.

$$\text{process}\ \frac{}{(\texttt{proc}; P, \delta) \xrightarrow{\rho}_{seq} (P; \texttt{proc}; P, \delta)}$$

$$\text{assign}\ \frac{v = [\![e]\!]\rho}{(\texttt{x <= } e; P, \delta) \xrightarrow{\rho}_{seq} (P, \delta[\texttt{x} \leftarrow v])}$$

$$\text{wait}\ \frac{}{(\texttt{wait on } W \texttt{ for } t; P, \delta) \xrightarrow{\rho}_{seq} (P, W, t, \delta)}$$

$$\text{true}\ \frac{[\![b]\!]\rho = \texttt{true}}{(\texttt{if } b \texttt{ then } P_1 \texttt{ else } P_2; P, \delta) \xrightarrow{\rho}_{seq} (P_1; P, \delta)}$$

$$\text{false}\ \frac{[\![b]\!]\rho = \texttt{false}}{(\texttt{if } b \texttt{ then } P_1 \texttt{ else } P_2; P, \delta) \xrightarrow{\rho}_{seq} (P_2; P, \delta)}$$

Fig. 2. Sequential execution of one process.

$$\Pi\text{-}i\ \frac{p_i \xrightarrow{\rho}_{seq} p_i'}{(p_1, \ldots, p_i, \ldots, p_n, \rho) \rightarrow (p_1, \ldots, p_i', \ldots p_n, \rho)}$$

$$\Delta\ \frac{\begin{array}{c} \forall i : p_i = (P_i, W_i, t_i, \delta_i) \qquad \rho' = \rho\delta_1 \ldots \delta_n \\ W = W_1 \cup \ldots \cup W_n \qquad E = \{x \mid \rho(x) \neq \rho'(x)\} \qquad E \cap W \neq \emptyset \\ p_i' = \begin{cases} (P_i, \delta_i) & \text{if } W_i \cap E \neq \emptyset \\ p_i & \text{otherwise} \end{cases} \end{array}}{(p_1, \ldots, p_n, \rho) \rightarrow (p_1', \ldots, p_n', \rho')}$$

$$\Theta\ \frac{\begin{array}{c} \forall i : p_i = (P_i, W_i, t_i, \delta_i) \qquad \rho' = \rho\delta_1 \ldots \delta_n \\ W = W_1 \cup \ldots \cup W_n \qquad E = \{x \mid \rho(x) \neq \rho'(x)\} \qquad E \cap W = \emptyset \\ \exists i : t_i \neq \infty \qquad t = min\{t_i \mid t_i \neq \infty\} \\ p_i' = \begin{cases} (P_i, \delta_i) & \text{if } t_i = t \\ (P_i, W_i, t_i - t, \delta_i) & \text{if } t_i \neq \infty \\ p_i & \text{otherwise} \end{cases} \end{array}}{(p_1, \ldots, p_n, \rho) \rightarrow (p_1', \ldots, p_n', \rho')}$$

where $\rho\delta$ is the function equal to δ on the domain of δ and to ρ otherwise.

Fig. 3. Simulation semantics.

case the process is sleeping, P_i is followed by the list W_i of signals it is sensitive to and the amount of time t_i which is left before the next timeout. Hence c_i has either the form P_i or (P_i, W_i, t_i). The driver δ_i stores the future value of signals as scheduled by the process. We let A_i denote the set of signals found on the left-hand side of some assignment statement appearing in the code of the process. A_i is the domain of δ_i. As indicated in Sect. 4.3.1.2 of [1], it is an error if

a signal has multiple sources. This imposes that processes assign distinct signals, so that drivers have disjoint domains. The previous restriction no longer holds for resolved signals. In this case, a user-defined function computes a value for the signal from its multiple sources. As stated before, resolution functions add no expressive power and we chose not to consider them. We finally let A be the set of signals that appear in some A_i.

Figure 2 depicts the sequential execution of an individual process in a global environment ρ. A process loops for ever, so the code which follows the loop command, is duplicated. When executing a signal assignment command, the expression on the right-hand side of the statement is evaluated in the current environment. The resulting value is not directly assigned in ρ to the target signal as would be the case in a more classical programming language. Rather, this action is delayed and the value is scheduled in the driver δ for the next synchronization point. Note that a subsequent assignment to the same signal can replace this value by another. The wait statement puts a process to sleep. The set of signals W is the sensitivity list and the integer t is the timeout. They both indicate the conditions under which the process will wake up: it will raise whenever one of the signals in W changes its value or the timeout expires. The rules for the if statement are the usual: the point where execution is transfered depends on the evaluation of the guard.

Then, the three rules in Fig. 3 suffice to completely characterize the simulation semantics of our kernel language. The first rule simply states that the processes all run concurrently in any possible order. The repetitive execution of this rule between the application of one of the next two rules is called a simulation cycle. When no process can run anymore, one of the rules Δ or Θ may apply. First, the global environment is updated. The values scheduled by processes for signals in their drivers are now taken into account. Recall that drivers have distinct domain. So, no conflict can occur between drivers with different update values for the same signal. Each of the two rules corresponds to a different waking up mechanism for the processes. The first rule is taken whenever an event has occurred on some of the signals in the sensitivity list of one of the processes. The execution of the awaken processes is resumed while the others stay still during the next simulation cycle. No physical time has elapsed. In the VHDL phraseology, the last simulation cycle is called a delta cycle. In the circuit analogy, this can be understood as a way to simulate the instantaneous propagation of electricity through wires (signals). This first rule has the priority over the next one which only applies when all signals in W remain unchanged. Then, the time is advanced until the smallest timeout has passed. This is a time-advance cycle. It may be that neither condition is fulfilled (all signals stay still and all processes sleep for the eternity), in which case execution is simply blocked.

In the following, we sometimes label the transition relation $\rightarrow$ by Π-i, Δ or Θ to identify which rule is used.

The semantics $[\![e]\!]\rho$ of an expression has nothing peculiar and is therefore not given. Let us just explain that the function `random` picks arbitrarily a value between `true` and `false`.

The initial configuration s_0 of a system is entirely determined by the code $Q_1 \mid \ldots \mid Q_n$ of the description and by an initial environment ρ_0:

$$s_0 = (p_1, \ldots, p_n, \rho_0)$$
$$\forall i \in [1, n] : p_i = (Q_i, \delta_i) \wedge \forall x \in A_i : \delta_i(x) = \rho_0(x).$$

Example 1. Let us observe the simulation of a small design. It includes only two processes. The first one models the environment:

```
-- this is a VHDL comment !
-- test harness.
proc;
CLK <= not CLK; I <= random; RST <= true;
wait for 10;
```

The environment generates a clock, writes arbitrary value into `I` and continuously drives the reset signal to `true`. The second component reacts to this environment:

```
-- unit under verification.
proc; wait on CLK;
if CLK then
    O <= I;
    if I then CNT <= CNT + 1; if RST then CNT <= 0;
```

This process is synchronized on the clock. It acts as a latch, that is, it propagates the value on its input `I` to its output `O` with a delay of one clock cycle. When the reset signal is low, it also counts the number of high occurrences on the input. Observe that, although most of the signals are booleans, the counter has the integer type. This simple design may thus have an infinite state space. We suppose the initial value of boolean and integer signals are respectively `false` and `0`. So, the initial environment and drivers are $\rho = [\texttt{CLK} \leftarrow \texttt{false}, \texttt{I} \leftarrow \texttt{false}, \texttt{RST} \leftarrow \texttt{false}, \texttt{O} \leftarrow \texttt{false}, \texttt{CNT} \leftarrow \texttt{0}]$, $\delta_1 = [\texttt{CLK} \leftarrow \texttt{false}, \texttt{I} \leftarrow \texttt{false}, \texttt{RST} \leftarrow \texttt{false}]$ and $\delta_2 = [\texttt{O} \leftarrow \texttt{false}, \texttt{CNT} \leftarrow \texttt{0}]$. In the following simulation of the two first cycles, let $P_2' = \texttt{if CLK then}\ldots$, $\delta_1' = [\texttt{CLK} \leftarrow \texttt{true}, \texttt{I} \leftarrow \texttt{true}, \texttt{RST} \leftarrow \texttt{true}]$ and $\delta_2' = [\texttt{O} \leftarrow \texttt{true}, \texttt{CNT} \leftarrow \texttt{0}]$.

$$(P_1, \delta_1, P_2, \delta_2, \rho) \xrightarrow{\Pi\text{-}1} (\texttt{CLK <= not CLK;}\ldots, \delta_1, P_2, \delta_2, \rho)$$
$$\xrightarrow{\Pi\text{-}1} (\texttt{I <= random;}\ldots, \delta_1[\texttt{CLK} \leftarrow \texttt{true}], P_2, \delta_2, \rho)$$
$$\xrightarrow{\Pi\text{-}1} (\texttt{RST <= true;}\ldots, \delta_1[\texttt{CLK} \leftarrow \texttt{true}, \texttt{I} \leftarrow \texttt{true}], P_2, \delta_2, \rho)$$
$$\xrightarrow{\Pi\text{-}1} (\texttt{wait for 10;}\ldots, \delta_1', P_2, \delta_2, \rho) \xrightarrow{\Pi\text{-}1} (P_1, \emptyset, 10, \delta_1', P_2, \delta_2, \rho)$$
$$\xrightarrow{\Pi\text{-}2}{}^{*} (P_1, \emptyset, 10, \delta_1', P_2', \{\texttt{CLK}\}, \infty, \delta_2, \rho) \xrightarrow{\Delta} (P_1, \emptyset, 10, \delta_1', P_2', \delta_2, \rho\delta_1')$$
$$\xrightarrow{\Pi\text{-}2}{}^{*} (P_1, \emptyset, 10, \delta_1', \texttt{if RST then CNT <= 0;}\ldots, [\texttt{O} \leftarrow \texttt{true}, \texttt{CNT} \leftarrow \texttt{1}], \rho\delta_1')$$
$$\xrightarrow{\Pi\text{-}2}{}^{*} (P_1, \emptyset, 10, \delta_1', P_2', \{\texttt{CLK}\}, \infty, \delta_2', \rho\delta_1') \xrightarrow{\Theta} (P_1, \delta_1', P_2', \{\texttt{CLK}\}, \infty, \delta_2', \rho\delta_1'\delta_2')$$

3 A Benign Parallelism

Now, imagine one wants to check that, in the previous example, the counter `CNT` is never incremented. To do so, one puts a third process in parallel with the two previous. This process continuously monitors the value of the counter and signals an error whenever it is incremented.

```
-- monitor.
proc; wait on CLK;
if CNT > 0 then ERR <= true;
```

Hence, we want to observe the value taken by `ERR` during any possible simulation of the parallel composition of the design together with the observer. In general, we are interested in computing the set of global environments generated by the simulation semantics:

$$\mathcal{O}_{\rightarrow} = \{\rho \mid \exists p_1, \ldots, p_n : s_0 \rightarrow^* (p_1 \ldots p_n, \rho)\}$$

To account for the concurrency in VHDL, we adopted an interleaving model (see the Π rule in Fig. 3). It can be computationally very expensive to fully explore all the interleaved executions of processes. Fortunately, this is not necessary: by applying a technique comparable to partial order reduction [9], we can shrink the state space to search. The global environment is modified only when all processes are stopped. Hence, for our purpose, it is sufficient to compute the states that can be observed at synchronization points during any simulation. These states are reached regardless of the particular order into which processes are run. Thus, we can safely pick a specific ordering and prune all the executions that do not follow it. Formally, this optimization is achieved by the mean of a function *choice*. It selects one among the processes that can be run from each configuration of the system. The rule for parallel execution is replaced by the more restrictive:

$$\Pi\text{-}i \frac{p_i \xrightarrow{\rho}_{seq} p_i' \qquad i = choice(p_1, \ldots p_n, \rho)}{(p_1, \ldots, p_i, \ldots, p_n, \rho) \rightarrow (p_1, \ldots p_i', \ldots p_n, \rho)}$$

In practice, we want to execute processes sequentially in a given order. We allow the execution of a process once all the preceding ones have already completed:

$$choice(p_1 \ldots p_n, \rho) = i \text{ such that } \begin{cases} p_i = (P_i, \delta_i) \\ \forall j < i : p_j = (P_j, W_j, t_j, \delta_j) \end{cases}$$

Of course, other definitions for *choice* may as well be proper. From now on, we denote by $\rightarrow_c$ the optimized operational semantics.

We consecrate the rest of this section to show that our transformation is correct. That is, the environments obtained during any run of the standard and the optimized versions of the simulation semantics are the same.

Property 1. Let ρ be an environment, then:

$$\exists p_1, \ldots, p_n : s_0 \rightarrow^* (p_1, \ldots, p_n, \rho) \Longleftrightarrow \exists p_1', \ldots, p_n' : s_0 \rightarrow_c^* (p_1', \ldots, p_n', \rho).$$

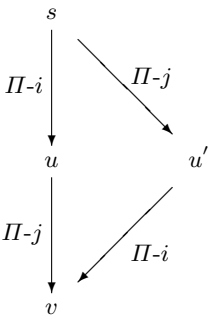


Fig. 4. Free swapping.

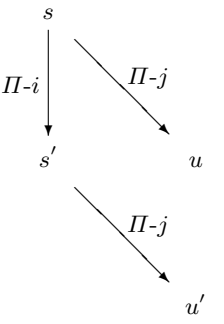


Fig. 5. Non blocking execution.

Proof. Since the optimized semantics is a restriction of the initial semantics, it can add no spurious states. The proof for the reciprocal is more demanding. We introduce $\multimap$ to represent the passage of a whole simulation cycle. It consists in the maximal execution of processes in parallel followed by one step of update:

Definition 1 (Simulation cycle). *Let the relation* $\multimap$ *be equal to* $\xrightarrow{\Pi}{}^{!}\xrightarrow{\Delta|\Theta}$, *where* $s \xrightarrow{\Pi}{}^{!} s' \iff s \xrightarrow{\Pi}{}^{*} s' \wedge \forall u : \neg(s \xrightarrow{\Pi} u)$.

A simple inspection of the transition rules reveals that the global environment is modified only at the end of each cycle. We immediately infer the following:

Lemma 1. *If* $s_0 \rightarrow^* (p_1, \ldots, p_n, \rho)$, *then there exists* $p'_1, \ldots, p'_n$ *such that* $s_0 \multimap^* (p'_1, \ldots, p'_n, \rho)$.

Picking a specific ordering in the execution of the processes still allow to attain the same configurations at the synchronization points:

Lemma 2. *If* $s \xrightarrow{\Pi}{}^{!} s'$ *then it is also true that* $s \xrightarrow{\Pi}{}^{!}_{c} s'$.

The proof is done by induction on the length of the derivation and can be found in appendix. On the contrary to [14], we can not rely on the determinism of VHDL to deduce the confluence of $\xrightarrow{\Pi}$. It appears that the `random` expression makes VHDL non-deterministic. Because of the `loop` command, it is also not nœtherian. Rather, we exploit the independence of processes in their concurrent execution. Thanks to the delayed assignment mechanism of signals, processes only interact during synchronization. This intuition is formalized by the two properties depicted in Fig. 4 and 5. First, the consecutive execution steps of two distinct processes can be freely swapped:

$$\forall s, u, v : s \xrightarrow{\Pi\text{-}i} u \wedge u \xrightarrow{\Pi\text{-}j} v \Rightarrow \exists u' : s \xrightarrow{\Pi\text{-}j} u' \wedge u' \xrightarrow{\Pi\text{-}i} v. \tag{1}$$

Second, performing one execution step of a process does not block the possible action of another:

$$\forall s, s', u : s \xrightarrow{\Pi\text{-}i} s' \wedge s \xrightarrow{\Pi\text{-}j} u \wedge i \not\equiv j \Rightarrow \exists u' : s' \xrightarrow{\Pi\text{-}j} u'. \tag{2}$$

We have just shown that $\mathcal{O}_{\rightarrow} \subseteq \mathcal{O}_{\multimap}$ (by lemma 1) and $\multimap \subseteq \multimap_c$ (implied by lemma 2). By definition 1, we have $\multimap_c^* \subseteq \rightarrow_c^*$. Transitivity allows us to conclude that, as expected, $\mathcal{O}_{\rightarrow} \subseteq \mathcal{O}_{\rightarrow_c}$.

4 A Generic Abstract Semantics

We have seen in the previous section, that, in order to check safety properties, we must estimate the values signals may take during any simulation of the design. We are thus interested in the set $\{s' \mid s_0 \rightarrow_c^* s'\}$ of states accessible from the initial configuration s_0 by a finite number of simulation steps. This is exactly the collecting semantics [11]. It can be expressed as the least fixpoint of the monotone operator $\mathbb{F}$:

$$\mathbb{F}(X) = \{s_0\} \cup \{s' \mid \exists s \in X : s \rightarrow_c s'\}$$

The fixpoint of $\mathbb{F}$ may be too large (even infinite) to be effectively computed. Following the methodology of abstract interpretation [10], we design an abstract domain in order to obtain a decidable, but approximate, description of the collecting semantics.

Thanks to a simple naming scheme, we can encode a tuple of global environment and drivers into a single function ϱ. The domain of ϱ is $V = S \cup \{x_\delta \mid \exists i : x \in A_i\}$. For every signal x in the design, it contains a variable x and possibly another variable x_δ to respectively denote the current and next value of the signal:

$$decode(\varrho) = (\rho, \delta_1, \ldots, \delta_n) \text{ such that } \begin{cases} \forall x \in S : \rho(x) = \varrho(x) \\ \forall x \in A_i : \delta_i(x) = \varrho(x_\delta) \end{cases}$$

We can decompose V into $V^{\mathbb{B}}$ and $V^{\mathbb{N}}$ according to the type of the signal corresponding to each variable. Then, ϱ is a function from $V^{\mathbb{B}}$ to booleans and from $V^{\mathbb{N}}$ to integers. The representation for sets of such mappings is left as a parameter of our abstract domain. We thus assume that we dispose of a complete lattice $(\mathcal{N}_V, \sqsubseteq, \sqcup, \bot, \sqcap, \top)$, which is related to $\wp(\mathbb{B}^{V^{\mathbb{B}}} \times \mathbb{N}^{V^{\mathbb{N}}})$ by a Galois connection $(\alpha_{\mathcal{N}}, \gamma_{\mathcal{N}})$. Our abstract domain is then the complete lattice $(\mathcal{L} \rightarrow \mathcal{N}, \sqsubseteq^\sharp, \sqcup^\sharp, \bot^\sharp, \sqcap^\sharp, \top^\sharp)$, with operations defined pointwise. An element X of this domain attaches to every combination of control points $(c_1, \ldots, c_n)$ in $\mathcal{L}$ an abstract representation, in $\mathcal{N}$, of tuples environment/drivers. Its meaning is expressed, in terms of sets of concrete configurations, thanks to the following complete meet morphism:

$$\gamma(X) = \left\{ (p_1, \ldots, p_n, \rho) \;\middle|\; \begin{array}{l} \varrho \in \gamma_{\mathcal{N}}(X(c_1, \ldots, c_n)) \wedge (\rho, \delta_1, \ldots, \delta_n) = decode(\varrho) \\ \forall i \in [1, n] : p_i = (c_i, \delta_i) \end{array} \right\}$$

The definition of the abstract transition relation depends on the existence of a few primitives of the numerical domain $\mathcal{N}$: an abstract assignment assign, an

$$\text{process}^\sharp \frac{}{(\texttt{proc}; P, R) \rightharpoonup_{seq} (P; \texttt{proc}; P, R)}$$

$$\text{assign}^\sharp \frac{R' = \mathsf{assign}_{\texttt{x}_\delta \leftarrow e}(R)}{(\texttt{x <= } e; P, R) \rightharpoonup_{seq} (P, R')}$$

$$\text{wait}^\sharp \frac{}{(\texttt{wait on } W \texttt{ for } t; P, R) \rightharpoonup_{seq} (P, W, t, R)}$$

$$\text{true}^\sharp \frac{R' = \mathsf{select}_b(R)}{(\texttt{if } b \texttt{ then } P_1 \texttt{ else } P_2; P, R) \rightharpoonup_{seq} (P_1; P, R')}$$

$$\text{false}^\sharp \frac{R' = \mathsf{select}_{\texttt{not } b}(R)}{(\texttt{if } b \texttt{ then } P_1 \texttt{ else } P_2; P, R) \rightharpoonup_{seq} (P_2; P, R')}$$

Fig. 6. Abstract sequential execution.

$$\Pi^\sharp\text{-}i \frac{(c_i, R) \rightharpoonup_{seq} (c_i', R') \qquad i \in \mathsf{choice}(c_1, \ldots, c_n, R)}{(c_1, \ldots, c_i, \ldots, c_n, R) \rightharpoonup (c_1, \ldots, c_i', \ldots c_n, R')}$$

$$\Delta^\sharp \frac{\begin{array}{c} \forall i : c_i = (P_i, W_i, t_i) \qquad W = W_1 \cup \ldots \cup W_n \\ E \in \mathsf{event}_W(R) \qquad E \neq \emptyset \qquad R' = \mathsf{update}_{E,W}(R) \\ c_i' = \begin{cases} P_i & \text{if } W_i \cap E \neq \emptyset \\ c_i & \text{otherwise} \end{cases} \end{array}}{(c_1, \ldots, c_n, R) \rightharpoonup (c_1', \ldots, c_n', R')}$$

$$\Theta^\sharp \frac{\begin{array}{c} \forall i : c_i = (P_i, W_i, t_i) \qquad W = W_1 \cup \ldots \cup W_n \\ \emptyset \in \mathsf{event}_W(R) \qquad R' = \mathsf{update}_{\emptyset,W}(R) \\ \exists i : t_i \neq \infty \qquad t = min\{t_i \mid t_i \neq \infty\} \\ c_i' = \begin{cases} P_i & \text{if } t_i = t \\ (P_i, W_i, t_i - t) & \text{if } t_i \neq \infty \\ c_i & \text{otherwise} \end{cases} \end{array}}{(c_1, \ldots, c_n, R) \rightharpoonup (c_1', \ldots, c_n', R')}$$

Fig. 7. Abstract simulation semantics.

abstract selection select to assert boolean conditions, a function event to gather the sets of modified signals, an abstract update to carry out the environment's modification that takes place during synchronization. These operations must check the following conditions:

$$
\begin{array}{ll}
\{\varrho' \mid \varrho \in \gamma_{\mathcal{N}}(R) \wedge v = [\![e]\!]\varrho \wedge \varrho' = \varrho[x \leftarrow v]\} & \subseteq \gamma_{\mathcal{N}}(\mathsf{assign}_{x \leftarrow e}(R)) \\
\{\varrho \mid \varrho \in \gamma_{\mathcal{N}}(R) \wedge [\![b]\!]\varrho = \mathtt{true}\} & \subseteq \gamma_{\mathcal{N}}(\mathsf{select}_b(R)) \\
\left\{ E \;\middle|\; \begin{array}{l} \varrho \in \gamma_{\mathcal{N}}(R) \\ E = \{x \in W \cap A \mid \varrho(x) \neq \varrho(x_\delta)\} \end{array} \right\} & \subseteq \gamma_{\mathcal{N}}(\mathsf{event}_W(R)) \\
\left\{ \varrho' \;\middle|\; \begin{array}{l} \varrho \in \gamma_{\mathcal{N}}(R) \\ \forall x \in W \cap A : \varrho(x) \neq \varrho(x_\delta) \Leftrightarrow x \in E \\ \varrho'(x) = \begin{cases} \varrho(x_\delta) & \text{if } x \in A \\ \varrho(x) & \text{otherwise} \end{cases} \end{array} \right\} & \subseteq \gamma_{\mathcal{N}}(\mathsf{update}_{E,W}(R)).
\end{array}
$$

At last, the abstract pruning procedure choice must have the property:

$$\{choice(s) \mid s \in \gamma(\{c_1, \ldots, c_n \mapsto R\})\} \subseteq \mathsf{choice}(c_1, \ldots, c_n, R).$$

The abstract transition relation $\rightharpoonup$, as given in Fig. 6 and 7, mimics the effects of the simulation semantics. It can be understood as the natural extension to set of states of the relation $\rightarrow$. We give the intuition for the abstract Δ rule. All the signals that are listened by any process are put in the set W. For each tuple environment/drivers that the abstract value R represents, the function event generates a set E. Elements of this set are signals in W for which the value scheduled in some driver is different from the current value in the environment. E entirely determines which processes need to be waked up for the next cycle. Thus, for every non-empty set E in $\mathsf{event}_W(R)$, the control points c_i are modified accordingly. The abstract environment is also updated.

The abstract representation X_0 for the initial state is:

$$
\begin{array}{rl}
X_0(c_1, \ldots, c_n) = & \begin{cases} \alpha_{\mathcal{N}}(\{\varrho_0\}) & \text{if } c_i = Q_i \\ \bot_{\mathcal{N}} & \text{otherwise} \end{cases} \\
\text{where } \forall x \in S \;:\; & \varrho_0(x) = \rho_0(x) \\
\forall x \in A \;:\; & \varrho_0(x_\delta) = \rho_0(x).
\end{array}
$$

We can now design the abstract counterpart of $\mathbb{F}$:

$$\mathbb{F}^\sharp(X)(c') = X_0(c') \sqcup \bigsqcup \{R' \mid \exists c : X(c) = R \wedge (c, R) \rightharpoonup (c', R')\}$$

Monotonicity of $\mathbb{F}^\sharp$ is readily checked. Besides, the properties enforced on the basic operators ensure the soundness $\mathbb{F}(\gamma(X)) \subseteq \gamma(\mathbb{F}^\sharp(X))$ of the abstract semantic function $\mathbb{F}^\sharp$. Consequently, it comes that the fixpoint of $\mathbb{F}^\sharp$ provides a sound upper-approximation of the collecting semantics:

Theorem 1 (Correction of the static analysis). $lfp_\emptyset \mathbb{F} \subseteq \gamma(lfp_{\bot^\sharp} \mathbb{F}^\sharp)$.

Kleene's theorem yields an algorithm to get the result of the static analysis. It suffices to compute the successive iterates $\bot^\sharp$, $\mathbb{F}^\sharp(\bot^\sharp)$, $\mathbb{F}^\sharp(\mathbb{F}^\sharp(\bot^\sharp))$,... of $\mathbb{F}^\sharp$ until stabilization. In practice, more optimized fixpoint computation algorithms [5,15] may be used. When the numerical domain $\mathcal{N}$ is not of finite height a widening [11] is required to force convergence.

5 The Domain of Constants and Partitioning

We instantiate the generic static analysis introduced in the previous section with a particular numerical domain. The literature about numerical domains is vast. Most [10,17,20,12] concern constraints over the integers. Some [6,19] represent boolean functions. Few [16] tackle the problem of mixing both booleans and integers. Much experiments need to be made in order to measure the fitness of each of these toward the verification of hardware design. As for now, let us take advantage of a very simple domain: the domain of constants $\mathcal{C}$. This domain is used in optimizing compilers [18] to suppress superfluous code. The abstract representation is a vector R that maps each variable x to an element $R(x)$ of one of the Hasse diagrams in Fig. 8 or 9. The corresponding meaning is:

$$\gamma_{\mathcal{C}}(\bot_{\mathcal{C}}) = \emptyset \qquad \gamma_{\mathcal{C}}(\top_{\mathcal{C}}^{\mathbb{B}}) = \mathbb{B} \qquad \gamma_{\mathcal{C}}(\top_{\mathcal{C}}^{\mathbb{N}}) = \mathbb{N}$$
$$\forall n \in \mathbb{N} : \gamma_{\mathcal{C}}(n) = \{n\} \qquad \gamma_{\mathcal{C}}(\texttt{true}) = \{\texttt{true}\} \qquad \gamma_{\mathcal{C}}(\texttt{false}) = \{\texttt{false}\}$$
$$\gamma_{\mathcal{C}_V}(R) = \{\varrho \mid \forall x \in V : \varrho(x) \in \gamma_{\mathcal{C}}(R[x])\}$$

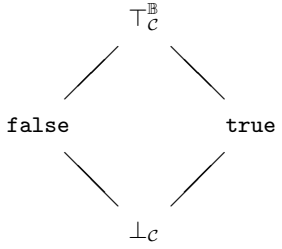


Fig. 8. Boolean constant domain.

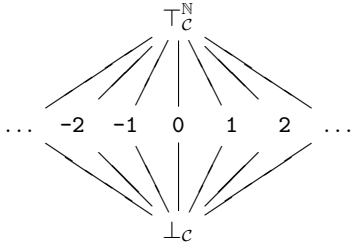


Fig. 9. Integer constant domain.

The lattice operations $\sqsubseteq_{\mathcal{C}}$, $\sqcup_{\mathcal{C}}$, and $\sqcap_{\mathcal{C}}$ on the domain of constants are entirely determined by the diagrams of Fig. 8 and 9. They are then extended pointwise to handle vectors on $\mathcal{C}_V$. This domain is non-relational in the sense that it forgets any kind of relationships between the values of variables. We can observe that, as soon as one of the component of R is equal to $\bot_{\mathcal{C}}$, then R represents the empty set of functions. The reduction operator [11] normalizes the representation. It smashes the bottom element of every components together:

$$\sigma(R)(x) = \begin{cases} \bot_{\mathcal{C}} & \text{if } \exists y : R(y) = \bot_{\mathcal{C}} \\ R(x) & \text{otherwise} \end{cases}$$

Letting d, *uop* and *bop* respectively range over the set of constant expressions $\{n, \texttt{true}, \texttt{false}\}$, unary operator $\{\texttt{not}\}$ and binary operators $\{\texttt{or}, \texttt{and}, \texttt{+}, \texttt{>}\}$, we can define the abstract evaluation of expressions thanks to the following rules:

$$
\begin{aligned}
[\![d]\!]_{\mathcal{C}} R &= d \\
[\![\mathtt{x}]\!]_{\mathcal{C}} R &= R(\mathtt{x}) \\
[\![\mathtt{random}]\!]_{\mathcal{C}} R &= \top_{\mathcal{C}}^{\mathbb{B}} \\
[\![uop\ e]\!]_{\mathcal{C}} R &= \overline{uop}\,([\![e]\!]_{\mathcal{C}} R) \\
[\![e_1\ bop\ e_2]\!]_{\mathcal{C}} R &= ([\![e_1]\!]_{\mathcal{C}} R)\ \overline{bop}\ ([\![e_2]\!]_{\mathcal{C}} R)
\end{aligned}
\qquad
v_1\ \overline{bop}\ v_2 = \begin{cases} \bot_{\mathcal{C}} & \text{if } v_1 = \bot_{\mathcal{C}} \text{ or } v_2 = \bot_{\mathcal{C}} \\ \top_{\mathcal{C}} & \text{if } v_1 = \top_{\mathcal{C}} \text{ or } v_2 = \top_{\mathcal{C}} \\ v_1\ bop\ v_2 & \text{otherwise} \end{cases}
$$

$$
\overline{uop}\ v = \begin{cases} \bot_{\mathcal{C}} & \text{if } v = \bot_{\mathcal{C}} \\ \top_{\mathcal{C}} & \text{if } v = \top_{\mathcal{C}} \\ uop\ v & \text{otherwise} \end{cases}
$$

Correct abstract primitives may be:

$$
\mathsf{assign}^{\mathcal{C}}_{x \leftarrow e}(R)(y) = \begin{cases} [\![e]\!]_{\mathcal{C}} R & \text{if } y = x \\ R(y) & \text{otherwise} \end{cases}
$$

$$
\mathsf{event}^{\mathcal{C}}_{W}(R) = \left\{ E \subseteq W \cap A \;\middle|\; \begin{cases} R(x) = d_1 \ \not= d_2 = R(x_\delta) \Rightarrow x \in E \\ R(x) = d = R(x_\delta) \Rightarrow x \notin E \end{cases} \right\}
$$

$$
\mathsf{update}^{\mathcal{C}}_{E,W}(R)(x) = \begin{cases} R(x) \sqcap_{\mathcal{C}} R(x_\delta) & \text{if } x \in A, x \in W \text{ and } x \notin E \\ R(x_\delta) & \text{if } x \in A \\ R(x) & \text{otherwise} \end{cases}
$$

In the case of the selection rule select_b, we assume that all the negations in the boolean expression b have been pushed down to the leaves. This is done using the two De Morgan laws $\neg(A \vee B) \iff (\neg A \wedge \neg B)$ and $\neg(A \wedge B) \iff (\neg A \vee \neg B)$. Then, we can have:

$$
\mathsf{select}^{\mathcal{C}}_{\mathtt{true}}(R)(y) = R(y) \qquad \mathsf{select}^{\mathcal{C}}_{\mathtt{x}}(R)(y) = \begin{cases} R(\mathtt{x}) \sqcap_{\mathcal{C}} \mathtt{true} & \text{if } y = \mathtt{x} \\ R(y) & \text{otherwise} \end{cases}
$$

$$
\mathsf{select}^{\mathcal{C}}_{\mathtt{false}}(R)(y) = \bot_{\mathcal{C}} \qquad \mathsf{select}^{\mathcal{C}}_{\mathtt{not\ x}}(R)(y) = \begin{cases} R(\mathtt{x}) \sqcap_{\mathcal{C}} \mathtt{false} & \text{if } y = \mathtt{x} \\ R(y) & \text{otherwise} \end{cases}
$$

$$
\mathsf{select}^{\mathcal{C}}_{e_1\ \mathtt{>}\ e_2}(R)(y) = \begin{cases} R(y) & \text{if } \mathtt{true} \sqsubseteq_{\mathcal{C}} [\![e_1\ \mathtt{>}\ e_2]\!]_{\mathcal{C}} R \\ \bot_{\mathcal{C}} & \text{otherwise} \end{cases}
$$

Although efficient, the constant approximation is too crude and leads to imprecise results. We refine our representation by partitioning [11] according to the values of the variables V_M of the monitor which asserts the property to check. That is, we now handle a function $R : \wp(\mathbb{B}^{V_M^{\mathbb{B}}} \times \mathbb{N}^{V_M^{\mathbb{N}}}) \to \mathcal{C}_V$ mapping each possible assignment of the variables in V_M to a vector of constants. We draw our inspiration from STE [8], where the property to check (a trajectory assertion) is marked by vectors of constants. Abstract primitives $\mathsf{assign}_{x \leftarrow e}$, select_b or $\mathsf{update}_{E,W}$ are all given by:

$$
\mathsf{f}(R)(a') = \bigsqcup_{\substack{\mathcal{C}_V \\ a \in dom(R)}} \mathsf{f}^{\mathcal{C}}(R(a)) \sqcap_{\mathcal{C}_V} a'
$$

As for the operator $\mathsf{event}_W(R)$, we have:

$$\mathsf{event}_W(R) = \bigcup_{a \in dom(R)} \mathsf{event}_W^{\mathcal{C}}(R(a))$$

In the following tables, T stands for `true`, F for `false`.

Table 1. Analyzing a correct design.

iteration n°	CLK	$\mathtt{CLK}_\delta$	CNT	$\mathtt{CNT}_\delta$	ERR	$\mathtt{ERR}_\delta$		RST	$\mathtt{RST}_\delta$	I	$\mathtt{I}_\delta$	O	$\mathtt{O}_\delta$
0	F	F	0	0	F	F	$\mapsto$	F	F	F	F	F	F
1	F	T	0	0	F	F	$\mapsto$	F	T	F	$\top$	F	F
2	T	T	0	0	F	F	$\mapsto$	T	T	$\top$	$\top$	F	$\top$
3	T	F	0	0	F	F	$\mapsto$	T	T	$\top$	$\top$	$\top$	$\top$
4	F	F	0	0	F	F	$\mapsto$	T	T	$\top$	$\top$	$\top$	$\top$
5	F	T	0	0	F	F	$\mapsto$	$\top$	T	$\top$	$\top$	$\top$	$\top$
6	T	T	0	0	F	F	$\mapsto$	T	T	$\top$	$\top$	$\top$	$\top$
7	T	F	0	0	F	F	$\mapsto$	T	T	$\top$	$\top$	$\top$	$\top$

Table 2. Analyzing a faulty design.

iteration n°	CLK	$\mathtt{CLK}_\delta$	CNT	$\mathtt{CNT}_\delta$	ERR	$\mathtt{ERR}_\delta$		RST	$\mathtt{RST}_\delta$	I	$\mathtt{I}_\delta$	O	$\mathtt{O}_\delta$
0	F	F	0	0	F	F	$\mapsto$	F	F	F	F	F	F
1	F	T	0	0	F	F	$\mapsto$	F	T	F	$\top$	F	F
2	T	T	0	0	F	F	$\mapsto$	T	T	$\top$	$\top$	F	$\top$
	T	T	0	1	F	F	$\mapsto$	T	T	$\top$	$\top$	F	$\top$
3	T	F	0	0	F	F	$\mapsto$	T	T	$\top$	$\top$	$\top$	$\top$
	T	F	1	1	F	F	$\mapsto$	T	T	$\top$	$\top$	$\top$	$\top$
4	F	F	0	0	F	F	$\mapsto$	T	T	$\top$	$\top$	$\top$	$\top$
	F	F	1	1	F	T	$\mapsto$	T	T	$\top$	$\top$	$\top$	$\top$
5	F	T	0	0	F	F	$\mapsto$	$\top$	T	$\top$	$\top$	$\top$	$\top$
	F	T	1	1	**T**	T	$\mapsto$	T	T	$\top$	$\top$	$\top$	$\top$

Note that in theory, as soon as one of the variables of the monitor is of integer type, then the function R is infinite. However, in practice, the representation can be kept sparse. Only the mappings of R with a non-bottom value need to be remembered. Hence, if the variables of the monitor may only take a finite number of values during any simulation of the system, then our analysis is sure to terminate. It is up to the user to write monitors that exhibit a finite behavior. It is worth noticing that the size of the abstract representation is only linear in the number of variables in the design under verification. It is thus possible to run this analysis on very big designs.

To be complete, we need to tell our pruning strategy is simply to run the processes sequentially in a given order:

$$\mathsf{choice}(c_1, \ldots, c_n, R) = \{i \mid p_i = (P_i, \delta_i) \wedge \forall j < i : p_j = (P_j, W_j, t_j, \delta_j)\}$$

Example 2. As an illustration, we let our analysis run on the code of example 1. The evolution of the abstract environment R at the synchronization point is shown in table 1. To make it more readable, we only write the elements that differ from the previous iteration. This example illustrates that, although the monitor's variable `CNT` has the integer type, the analysis terminates.

Now, suppose the two last commands of the unit process are swapped. This gives the erroneous design:

```
-- erroneous unit.
proc; wait on CLK;
if CLK then
    O <= I;
    if RST then CNT <= 0; if I then CNT <= CNT + 1;
```

Whenever the signal `I` is high, `CNT` is set to `0` by the command "`if RST then CNT <= 0;`". However the next command immediately increments it. This is not the behavior expected by the monitor. In fact, the analysis correctly finds out, see table 2, that the `ERR` signal may become `true`.

6 Conclusion

We have presented a static analysis for behavioral VHDL. It automatically computes an approximate description of the states that are reachable during any simulation of the design. One can use this information to check safety properties that are expressed as observers. The algorithm is derived from a concise operational semantics of VHDL, which pinpoints the essential concepts of hardware simulation. We showed how benign the VHDL's form of parallelism is. We take advantage of this result by reducing the state space to explore. Our algorithm is generic in the underlying representation of sets of states. Hence, trade-offs between memory usage and precision can be made by plugging in different abstract numerical domains.

The ability of our analysis to handle behavioral descriptions with infinite data-types, makes it applicable at a very early stage in the design cycle. The fact that no distinct formalism is necessary to express the properties, makes the technique non-intrusive. Even if the algorithm fails, monitors can be reused for simulation or with other verification methods.

More work needs to be made in order to find adequate abstract numerical domains. It may be particularly effective to dynamically adjust the level of approximation, as in [4]. One can also study the conception of very efficient domains tailored at the verification of a specific class of properties. These could work hand-in-hand with intuitive logics that document a specific aspect of the system.

Acknowledgments. We would like to thank R. Pacalet for his patient explanations about the field of hardware design, R. Cousot, J. Feret and X. Rival for their advice, comments and technical help.

References

1. ANSI/IEEE Std 1076-1987. *IEEE Standard VHDL Language Reference Manual*, 1988.
2. IEEE Synthesis Interoperability W.G. 1076.6. Draft standard for VHDL synthesis subset level 2. http://www.eda.org/siwg.
3. A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, 1999.
4. F. Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4), 1992.
5. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the Int. Conf. on Formal Methods in Programming and their Applications*, volume 735 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
6. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *ieeetc*, C-35(8), 1986.
7. J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4), 1994.
8. C.-T. Chou. The mathematical foundation of symbolic trajectory evaluation. *Lecture Notes in Computer Science*, 1633, 1999.
9. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*. ACM Press, 1977.
11. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13, 1992. (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.dmi.ens.fr/~cousot`.).
12. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, 1978.
13. P. Georgelin, V. Rodrigues, and D. Borrione. An ACL2 model of VHDL for symbolic simulation and formal verification. In *XIII Symposium on Integrated Circuits and Systems Design (SBCCI'00)*, 2000.
14. K. Goossens. Reasoning about VHDL using operational and observational semantics. In *Correct Hardware Design and Verification Methods, CHARME*, volume 987 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.
15. S. Horwitz, A.J. Demers, and T. Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.
16. B. Jeannet. Dynamic partitioning in linear relation analysis. Technical report, BRICS, 2000.
17. Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6, 1976.
18. G. Kildall. A unified approach to global program optimisation. In *ACM Symposium on Principles of Programming Languages*, 1973.
19. Laurent Mauborgne. Abstract interpretation using TDGs. In *Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
20. A. Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO II*, volume 2053 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

21. W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl. The simulation semantics of SystemC. www.systemc.org.
22. Open Verilog International (OVI), 15466 Los Gatos Boulevard, Suite 109-071, Los Gatos, CA 95032. *Verilog HDL Language Reference Manual.*
23. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.
24. C-J.H. Seger and R.E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design: An International Journal*, 6(2), 1995.

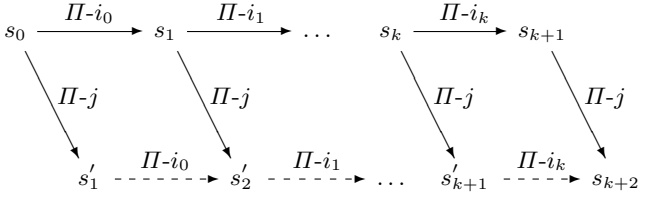
A Proof of Lemma 2

We have to show that if $s \xrightarrow{\Pi}{}^{!} s'$ then $s \xrightarrow{\Pi}{}^{!}_{c} s'$. This is proved by induction on the length of the derivation of $\xrightarrow{\Pi}$ from s to s'. The base case is trivial, since s is terminal and $s = s'$. For the induction step, consider the derivation $s_0 \xrightarrow{\Pi\text{-}i_0} s_1 \ldots \xrightarrow{\Pi\text{-}i_n} s_{n+1}$. Let us define $j = \mathit{choice}(s)$. Now, let k be the smallest integer such that $i_k = j$. It necessarily exists. Indeed, if it doesn't then repetitive application of the property (2) allows us to establish that there is s_{n+2} and $s_{n+1} \xrightarrow{\Pi\text{-}j} s_{n+2}$. This is clearly impossible, since s_{n+1} is terminal. If we can find a derivation of $\xrightarrow{\Pi}$ from s'_1 to s_{k+1}, we can apply the induction hypothesis and conclude that $s_0 \xrightarrow{\Pi}{}^{!}_{c} s'_{n+1}$.

To build this derivation, the idea is to exchange the Π-j step between state s_k and s_{k+1} with the one occurring between s_0 and s'_1. This is done by degrees thanks to the property (1). We prove by induction on k that if:

$$\begin{cases} s_0 \xrightarrow{\Pi\text{-}i_0} s_1 \ldots \xrightarrow{\Pi\text{-}i_{k-1}} s_k \wedge \forall l \in [0, k[: i_l \neq j \\ s_k \xrightarrow{\Pi\text{-}j} s_{k+1} \end{cases}$$

then there exists $s_0 \xrightarrow{\Pi\text{-}j} s'_1 \xrightarrow{\Pi\text{-}i_0} s'_2 \ldots \xrightarrow{\Pi\text{-}i_{k-1}} s_{k+1}$. This is trivial for the base case. For the induction step, by property (1), we can perform a swapping and it comes that there is s'_{k+1} such that $s_k \xrightarrow{\Pi\text{-}j} s'_{k+1} \xrightarrow{\Pi\text{-}i_k} s_{k+2}$. We can then apply the induction hypothesis to complete the construction. This argument is pictured below:



States vs. Traces in Model Checking by Abstract Interpretation

Roberto Giacobazzi[1] and Francesco Ranzato[2]

[1] Dipartimento di Informatica
Università di Verona
Strada Le Grazie 15, 37134 Verona, Italy
giaco@sci.univr.it

[2] Dipartimento di Matematica Pura ed Applicata
Università di Padova
Via Belzoni 7, 35131 Padova, Italy
franz@math.unipd.it

Abstract. In POPL'00, Cousot and Cousot showed that the classical *state-based* model checking of a very general temporal language called $\overset{\curvearrowright}{\mu}$-calculus is an *incomplete* abstract interpretation of its *trace-based* semantics. In ESOP'01, Ranzato showed that the least refinement of the state-based model checking semantics of the $\overset{\curvearrowright}{\mu}$-calculus which is *complete* w.r.t. its trace-based semantics exists, and it is essentially the trace-based semantics itself. The analogous problem in the opposite direction is solved by the present paper. First, relatively to any incomplete temporal connective of the $\overset{\curvearrowright}{\mu}$-calculus, we characterize the structure of the models, i.e. transition systems, for which the state-based model checking is trace-complete. On this basis, we prove that the unique abstraction of the state-based model checking semantics of the $\overset{\curvearrowright}{\mu}$-calculus (actually, of any fragment allowing conjunctions) which is complete w.r.t. the trace-based semantics is the straightforward semantics carrying no information at all. The following consequence can be drawn: there is no way to either refine or abstract sets of states in order to get a model checking algorithm for (any fragment allowing conjunctions of) the $\overset{\curvearrowright}{\mu}$-calculus which is trace-complete.

1 Introduction

The standard state-based model checking problem consists in characterizing the set of all the states of a transition system, modelling some reactive system, that satisfy a given temporal specification ϕ. ϕ is defined within some temporal logic language and specifies the required temporal properties of the system to be verified [7,16]. This model checking procedure must collect all the states s of the model M such that all the possible executions (in the existential checking: "there exists one execution") in M departing from s satisfy ϕ: $\{s \in \mathit{State} \mid M, s \models_{state} \phi\}$, where $\models_{state}$ is used to emphasize the state-based semantics of temporal formulae (i.e., the semantics of a temporal formula is a set of states). Although model checking systems are state-based, i.e. they solve

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 461–476, 2002.
© Springer-Verlag Berlin Heidelberg 2002

the above state-based model checking problem, semantics of temporal calculi can be also naturally given in terms of sets of execution traces, i.e. sequences of states distributed along a discrete time-line. Obviously, traces are much richer than states, and, as advocated by Cousot and Cousot [11] in their approach to model checking, the natural semantics of a temporal formula should be a set of execution traces. Nevertheless, for obvious pragmatic reasons of efficiency, model-checkers handle and observe states only, i.e., they abstract away from traces observing states. This abstraction is the subject of this paper. A far more concrete model checking problem can then be formulated for traces: the semantics of a temporal formula is a set of traces, and $M, s \models_{trace} \phi$ means that all the traces in M with present state s satisfy ϕ, i.e. belong to the semantics of ϕ. It should be clear that trace-based model checking is strictly more precise, but it is obviously unfeasible to design a practical model checking algorithm for system verification handling sets of traces. It is not clear however whether it is possible to find some approximation of the trace-based model checking problem which can be solved starting from the state-based model checking with no loss of precision, namely this approximate checking should be logically equivalent to the trace-based checking. More precisely, the paper answers the following question: is it possible to minimally refine or abstract the state-based semantics of a general temporal calculus so that this induces a corresponding model checking which is trace-complete, i.e. logically equivalent to the trace-based model checking? In our approach, refinements and abstractions of a semantics are intended to be expressed by standard abstract interpretation. As far as refinements are concerned, the negative answer has been given by Ranzato [17]: the only semantic refinement of the state-based semantics inducing a trace-complete model checking is the trace-based semantics itself. This paper faces with the remaining question and also in this case reports a negative answer: there exists no trace-complete abstraction of the state-based semantics but for the trivial semantics carrying no information at all. Thus, summing up, this result shows that states are, so to say, "intrinsically trace-incomplete", since there is no way to get a trace-complete model checking by modifying by refinement or by abstraction the state-based semantics.

The Scenario. This result is formulated and shown within the Cousot and Cousot's [11] abstract interpretation approach to model checking. In POPL'00, Cousot and Cousot proposed a general framework, called *temporal abstract interpretation*, introducing an enhanced temporal calculus, called $\overset{\curvearrowright}{\mu}$-calculus and inspired by Kozen's μ-calculus, with a trace-based semantics, and hence featuring a trace-based model checking. The state-based model checking is then specified as an abstract interpretation of the trace-based model checking. The trace-based semantics of the $\overset{\curvearrowright}{\mu}$-calculus is time-symmetric: this means that execution traces have potentially infinite length both in the future and in the past. This time-symmetry is not the only new feature of the $\overset{\curvearrowright}{\mu}$-calculus. The $\overset{\curvearrowright}{\mu}$-calculus also provides a tight combination of linear- and branching-time, allowing to derive classical specification languages like CTL, CTL*, and Kozen's μ-calculus itself, as suitable fragments.

One of the main achievements of Cousot and Cousot's paper [11] is that state-based model checking has been reduced to an abstract interpretation of the trace-based semantics of the $\overset{\curvearrowright}{\mu}$-calculus. The semantics $[\![\phi]\!]^{trace}$ of a temporal specification ϕ is the set of traces of a transition system M making ϕ true. Coherently with the state-based model checking problem, the state-abstraction $\alpha_M^{\forall}$ abstracts a set of traces $[\![\phi]\!]^{trace}$ to the set of states of M such that any execution of M departing from s satisfies the formula ϕ: $\alpha_M^{\forall}([\![\phi]\!]^{trace}) = \{s \in State \mid M, s \models_{trace} \phi\}$, where $M, s \models_{trace} \phi$ means that all the traces in M with present state s are in $[\![\phi]\!]^{trace}$. This abstraction $\alpha_M^{\forall}$ induces a state-based semantics $[\![\cdot]\!]^{state}$ for the $\overset{\curvearrowright}{\mu}$-calculus which is sound by construction with respect to the trace semantics: for any temporal formula ϕ, $[\![\phi]\!]^{state} \subseteq \alpha_M^{\forall}([\![\phi]\!]^{trace})$. As proved in [11], this inclusion may be strict, namely the state-based model checking of the $\overset{\curvearrowright}{\mu}$-calculus is trace-incomplete. This means that there exists some formula ϕ and state s of the system such that $M, s \models_{trace} \phi$ (viz., $s \in \alpha_M^{\forall}([\![\phi]\!]^{trace})$), while $M, s \not\models_{state} \phi$ (viz., $s \not\in [\![\phi]\!]^{state}$). This incompleteness means that the semantics $[\![\cdot]\!]^{state}$ used for designing and proving preserving or even strongly preserving properties of most state-based model checking algorithms is incomplete w.r.t. traces. The same holds even for abstract model checking [5,7,12,15], where the abstraction map actually is a state-abstraction and can be modeled as a further abstract interpretation step of $[\![\cdot]\!]^{state}$ [11,14]. It is therefore important in order to understand the limits of state-based (concrete or abstract) model checking with respect to properties of traces, to investigate whether it is possible to find a semantics $[\![\cdot]\!]^{?}$ as a refinement or abstraction of $[\![\cdot]\!]^{state}$ which is complete for the trace-based semantics $[\![\cdot]\!]^{trace}$.

The Main Result. Ranzato [17] proved that any refinement of the state-based model checking, i.e. which can be obtained by refining the abstraction $\alpha_M^{\forall}$, is still incomplete with respect to the trace-based semantics of the full $\overset{\curvearrowright}{\mu}$-calculus. This means that the most abstract semantics which is trace-complete and includes the state-based semantics, is the trace-based semantics itself. This shows that there is no way to enhance state-based model checking to get completeness for traces unless having traces themselves in the model. In this paper we consider the symmetrical situation: instead of refining the state-based model checking abstraction $\alpha_M^{\forall}$, we are interested in isolating those abstractions of $\alpha_M^{\forall}$ which are trace-complete, namely no loss of precision is introduced in the verification by using a state-based model checker with respect to check the same property on the trace-based semantics.

The fact that completeness can be achieved not only by refining abstract domains but also by abstracting them should not be surprising. In this case the abstraction is intended to remove from the abstract domain the source of potential incompleteness in such a way that the resulting domain contains the largest amount of information in order to let completeness be achieved. Consider for instance the abstract domain $Sign^+ = \{\mathbb{Z}, [0, +\infty], [-\infty, 0], [0, 10], [0]\}$. This domain is not complete for integer multiplication: for example, 2×2 is approximated in $Sign^+$ by $[0, 10]$ while the abstract multiplication $\otimes$ in $Sign^+$ of the same expression $[0, 10] \otimes [0, 10]$ is $[0 + \infty]$. However, $Sign = \{\mathbb{Z}, [0, +\infty], [-\infty, 0], [0]\}$,

which is an abstraction of $Sign^+$, turns out to be complete for multiplication, i.e. abstract multiplication can be performed with no loss of precision with respect to the approximation of the concrete multiplication. *Sign* is the most concrete domain which abstracts $Sign^+$ and it is complete for multiplication: this is called the *core* of $Sign^+$ (see [13] for details). The core of $Sign^+$ removed the abstract value $[0, 10]$, which was the unique source of incompleteness.

We first characterize the structure of transition systems for which the state-based model checking is complete for the basic modalities predecessor, time-reversal, and conjunction of the $\overset{\curvearrowright}{\mu}$-calculus. In particular, conjunction turns out to be the crucial connective: in fact, the core of the state-based model checking for the conjunction is the straightforward abstraction of states carrying no information at all. On the basis of this fact, we prove that, for any fragment of the $\overset{\curvearrowright}{\mu}$-calculus allowing arbitrary conjunctions, the straightforward abstraction is the unique abstraction of the state-based model checking which induces a corresponding model checking which is complete for the trace-based semantics. This result, together with the one proved by Ranzato [17], shows that there is no way to get a complete approximation of the trace-based semantics by either refining or approximating the state-based model checking, emphasizing the intrinsic limits of the precision of state-based model checking with respect to the trace-based semantics of the $\overset{\curvearrowright}{\mu}$-calculus. This proves that state-based model checking cannot be used as an economic way to prove properties of traces in a complete way. In particular, since abstract model checking can be viewed as abstract interpretation of $[\![\cdot]\!]^{state}$ (cf. [11]), this also implies that there is no abstraction α of states (unless it is the straightforward abstraction) such that no loss of precision occurs by considering the abstract model checking $[\![\cdot]\!]^{\alpha(state)}$ with respect to $\alpha([\![\cdot]\!]^{trace})$. Otherwise stated, any abstract model checking is intrinsically incomplete with respect to the trace-based semantics of the $\overset{\curvearrowright}{\mu}$-calculus.

2 Abstract Interpretation and Model Checking

2.1 Abstract Interpretation Basics

The structure $\langle \mathrm{uco}(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top, \lambda x.x\rangle$ denotes the complete lattice of closure operators on a complete lattice $\langle C, \leq, \vee, \wedge, \top, \bot\rangle$, where $\rho \sqsubseteq \eta$ iff $\forall x \in C.\ \rho(x) \leq \eta(x)$. Throughout the paper, for any $\rho \in \mathrm{uco}(C)$, we follow a standard notation by denoting the image $\rho(C)$ simply by ρ itself: This does not give rise to ambiguity, since one can readily distinguish the use of ρ as function or set according to the context. Let us recall that (i) each closure $\rho \in \mathrm{uco}(C)$ is uniquely determined by the set of its fix-points, which coincides with its image, i.e. $\rho = \{x \in C \mid \rho(x) = x\}$ and (ii) $\rho \sqsubseteq \eta$ iff $\eta \subseteq \rho$.

Within the standard Cousot and Cousot framework, abstract domains can be equivalently specified either by Galois connections/insertions (GCs/GIs) or by closure operators [10]. In the first case, concrete and abstract domains C and A — for simplicity, these are assumed to be complete lattices — are related by a pair of adjoint functions $\alpha : C \to A$ and $\gamma : A \to C$, compactly denoted by (α, C, A, γ), and therefore C and A may consist of objects having different representations.

In the second case, instead, an abstract domain is specified as a closure operator on the concrete domain C (this closure could be also given by means of its set of fix-points). Given a concrete domain C, we will identify $\mathrm{uco}(C)$ with the so-called complete lattice $\mathcal{L}_C$ of abstract interpretations of C (cf. [9, Section 7] and [10, Section 8]). The ordering on $\mathrm{uco}(C)$ corresponds precisely to the standard order used in abstract interpretation to compare abstract domains with regard to their precision: A_1 is more precise (or concrete) than A_2 iff $A_1 \sqsubseteq A_2$ in $\mathrm{uco}(C)$. Thus, lub's $\sqcup$ and glb's $\sqcap$ on $\mathcal{L}_C$ give, respectively, the most precise abstraction and the most abstract concretization of a family of abstract domains.

Complete Abstract Interpretations. Let us succinctly recall the basic notions concerning completeness in abstract interpretation. Let $f : C \to C$ be a monotone or antitone concrete semantic function[1] occurring in some complex semantic specification, and let $f^\sharp : A \to A$ be a corresponding abstract function, where $A \in \mathcal{L}_C$. Then, $\langle A, f^\sharp \rangle$ is a sound abstract interpretation — or $f^\sharp$ is a correct approximation of f relatively to A — when $\forall c \in C.\ \alpha(f(c)) \leq_A f^\sharp(\alpha(c))$. On the other hand, $\langle A, f^\sharp \rangle$ is complete when equality holds, i.e. $\alpha \circ f = f^\sharp \circ \alpha$. Thus, completeness means that abstract computations accumulate no loss of information.

Any abstract domain $A \in \mathcal{L}_C$ induces the so-called canonical best correct approximation $f^A : A \to A$ of $f : C \to C$, defined by $f^A \stackrel{\mathrm{def}}{=} \alpha \circ f \circ \gamma$. This terminology is justified by the fact that any $f^\sharp : A \to A$ is a correct approximation of f iff $f^A \sqsubseteq f^\sharp$. Consequently, any abstract domain always induces an (automatically) sound abstract interpretation. However, not all abstract domains induce a complete abstract interpretation. It turns out that whenever a complete abstract function exists then this actually is the best correct approximation. This therefore means that completeness for an abstract function is a property which depends on the underlying abstract domain only. Thus, for abstract domains specified by closure operators, an abstract domain $\rho \in \mathcal{L}_C$ is defined to be complete for f if $\rho \circ f = \rho \circ f \circ \rho$. More in general, this definition of completeness can be naturally extended to a set F of semantic functions by requiring completeness for each $f \in F$. Throughout the paper, we will adopt the following notation: $\Gamma(C, f) \stackrel{\mathrm{def}}{=} \{\rho \in \mathcal{L}_C \mid \rho \text{ is complete for } f\}$. Hence, for a set F, $\Gamma(C, F) = \cap_{f \in F} \Gamma(C, f)$.

2.2 Temporal Abstract Interpretation

Let us recall the basic notions and definitions of Cousot and Cousot's [11] abstract interpretation-based approach to model checking. $\mathbb{S}$ is a given, possibly infinite, set of states. Discrete time is modeled by the whole set of integers and therefore paths of states are time-symmetric, in particular are infinite also in the past. Thus, $\mathbb{P} \stackrel{\mathrm{def}}{=} \mathbb{Z} \to \mathbb{S}$ is the set of paths (an execution path with an initial state s can then be encoded by repeating forever in the past the state s). Of course,

[1] For simplicity, we consider unary functions with the same domain and co-domain, since the extension to the general case is conceptually straightforward.

traces keep track of the present time, and hence $\mathbb{T} \stackrel{\text{def}}{=} \mathbb{Z} \times \mathbb{P}$ is the set of traces. A (temporal) model is simply a set of traces: $\mathbb{M} \stackrel{\text{def}}{=} \wp(\mathbb{T})$ is the set of temporal models. The semantics of a temporal logical formula ϕ will be a temporal model, that will be the set of all and only those traces making ϕ true.

Models to check will be generated by transition systems, encoding some reactive system. The transition relation $\rightarrow \subseteq \mathbb{S} \times \mathbb{S}$ is assumed to be total, i.e., $\forall s \in \mathbb{S}. \exists s' \in \mathbb{S}.\, s \rightarrow s'$ and $\forall s' \in \mathbb{S}. \exists s \in \mathbb{S}.\, s \rightarrow s'$. This is not restrictive, since any transition relation can be lifted to a total transition relation simply by adding transitions $s \rightarrow s$ for any state s which is not reachable or which cannot reach any state. The model generated by the transition system $\langle \mathbb{S}, \rightarrow \rangle$ is therefore defined as $\mathcal{M}_{\rightarrow} \stackrel{\text{def}}{=} \{\langle i, \sigma \rangle \in \mathbb{T} \mid i \in \mathbb{Z},\ \forall k \in \mathbb{Z}.\, \sigma_k \rightarrow \sigma_{k+1}\}$.

The reversible $\overset{\curvearrowleft}{\mu}$-calculus has been introduced by Cousot and Cousot [11] as a generalization of Kozen's μ-calculus, provided with a trace-based semantics. Throughout the paper, $\mathbb{X}$ will denote an infinite set of logical variables.

Definition 2.1 ([11, Definition 13]). Formulae ϕ of the reversible $\overset{\curvearrowleft}{\mu}$-calculus are inductively defined as follows:

$$\phi ::= \boldsymbol{\sigma}_S \mid \boldsymbol{\pi}_t \mid X \mid \oplus\, \phi \mid \phi^{\curvearrowleft} \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \boldsymbol{\mu} X.\phi \mid \boldsymbol{\nu} X.\phi \mid \forall \phi_1 : \phi_2$$

where the quantifications are as follows: $S \in \wp(\mathbb{S})$, $t \in \wp(\mathbb{S} \times \mathbb{S})$, and $X \in \mathbb{X}$. $\mathfrak{L}_{\overset{\curvearrowleft}{\mu}}$ denotes the set of $\overset{\curvearrowleft}{\mu}$-calculus formulae. □

The intuition is that a closed formula ϕ is interpreted as the set of traces which make ϕ true. The trace-based semantics for the $\overset{\curvearrowleft}{\mu}$-calculus goes as follows.

Definition 2.2 ([11, Definition 13]). $\mathbb{E} \stackrel{\text{def}}{=} \mathbb{X} \rightarrow \mathbb{M}$ is the set of environments over $\mathbb{X}$. Given $\xi \in \mathbb{E}$, $X \in \mathbb{X}$ and $N \in \mathbb{M}$, $\xi[X/N] \in \mathbb{E}$ is defined to be the environment acting as ξ in $\mathbb{X} \smallsetminus \{X\}$ and mapping X to N. The $\overset{\curvearrowleft}{\mu}$-calculus semantics $[\![\cdot]\!] : \mathfrak{L}_{\overset{\curvearrowleft}{\mu}} \rightarrow \mathbb{E} \rightarrow \mathbb{M}$ is inductively and partially (because least or greatest fix-points could not exist) defined as follows:

$$\begin{array}{ll}
[\![\boldsymbol{\sigma}_S]\!]\xi \stackrel{\text{def}}{=} \boldsymbol{\sigma}_{\{\!\{S\}\!\}} & [\![\phi_1 \vee \phi_2]\!]\xi \stackrel{\text{def}}{=} [\![\phi_1]\!]\xi \cup [\![\phi_2]\!]\xi \\
[\![\boldsymbol{\pi}_t]\!]\xi \stackrel{\text{def}}{=} \boldsymbol{\pi}_{\{\!\{t\}\!\}} & [\![\neg\phi]\!]\xi \stackrel{\text{def}}{=} \neg([\![\phi]\!]\xi) \\
[\![X]\!]\xi \stackrel{\text{def}}{=} \xi(X) & [\![\boldsymbol{\mu} X.\phi]\!]\xi \stackrel{\text{def}}{=} \mathit{lfp}(\lambda N \in \mathbb{M}.[\![\phi]\!]\xi[X/N]) \\
[\![\oplus\phi]\!]\xi \stackrel{\text{def}}{=} \oplus([\![\phi]\!]\xi) & [\![\boldsymbol{\nu} X.\phi]\!]\xi \stackrel{\text{def}}{=} \mathit{gfp}(\lambda N \in \mathbb{M}.[\![\phi]\!]\xi[X/N]) \\
[\![\phi^{\curvearrowleft}]\!]\xi \stackrel{\text{def}}{=} {}^{\curvearrowleft}([\![\phi]\!]\xi) & [\![\forall\phi_1 : \phi_2]\!]\xi \stackrel{\text{def}}{=} \forall([\![\phi_1]\!]\xi, [\![\phi_2]\!]\xi),
\end{array}$$

where the corresponding temporal model transformers are defined as follows:

- For any $S \in \wp(\mathbb{S})$, $\boldsymbol{\sigma}_{\{\!\{S\}\!\}} \stackrel{\text{def}}{=} \{\langle i, \sigma \rangle \in \mathbb{T} \mid \sigma_i \in S\} \in \mathbb{M}$ is the S-state model, i.e., the set of traces whose current state is in S.
- For any $t \in \wp(\mathbb{S} \times \mathbb{S})$, $\boldsymbol{\pi}_{\{\!\{t\}\!\}} \stackrel{\text{def}}{=} \{\langle i, \sigma \rangle \in \mathbb{T} \mid (\sigma_i, \sigma_{i+1}) \in t\} \in \mathbb{M}$ is the t-transition model, i.e., the set of traces whose next step is a t-transition.
- $\oplus : \mathbb{M} \rightarrow \mathbb{M}$ is the predecessor transformer:
  $\oplus(X) \stackrel{\text{def}}{=} \{\langle i-1, \sigma \rangle \in \mathbb{T} \mid \langle i, \sigma \rangle \in X\} = \{\langle i, \sigma \rangle \in \mathbb{T} \mid \langle i+1, \sigma \rangle \in X\}$.

– $\curvearrowleft : \mathbb{M} \to \mathbb{M}$ is the reversal transformer:
$\curvearrowleft(X) \stackrel{\text{def}}{=} \{\langle -i, \lambda k.\sigma_{-k}\rangle \in \mathbb{T} \mid \langle i, \sigma\rangle \in X\}$.
– $\neg : \mathbb{M} \to \mathbb{M}$ is the complement transformer:
$\neg X \stackrel{\text{def}}{=} \mathbb{M} \smallsetminus X$.
– Given $s \in \mathbb{S}$, $(\cdot)_{\downarrow s} : \mathbb{M} \to \mathbb{M}$ is the state projection operator:
$X_{\downarrow s} \stackrel{\text{def}}{=} \{\langle i, \sigma\rangle \in X \mid \sigma_i = s\}$.
– $\forall : \mathbb{M} \times \mathbb{M} \to \mathbb{M}$ is the universal state closure transformer:
$\forall(X, Y) \stackrel{\text{def}}{=} \{\langle i, \sigma\rangle \in X \mid X_{\downarrow \sigma_i} \subseteq Y\}$. □

The successor operator $\ominus$ on traces can be dually defined as follows:

$$\ominus(X) \stackrel{\text{def}}{=} \{\langle i+1, \sigma\rangle \in \mathbb{T} \mid \langle i, \sigma\rangle \in X\} = \{\langle i, \sigma\rangle \in \mathbb{T} \mid \langle i-1, \sigma\rangle \in X\}.$$

Within this trace-based framework the model checking problem is easily formulated as follows. A closed temporal specification $\phi \in \mathfrak{L}_{\widehat{\mu}}$ is identified by its semantics, namely by the temporal model $[\![\phi]\!]\varnothing \in \mathbb{M}$. Thus, the *universal model checking* of a system $\mathcal{M}_{\to}$ against a specification ϕ amounts to check whether $\mathcal{M}_{\to} \subseteq [\![\phi]\!]\varnothing$. Dually, in the *existential model checking* the goal is that of checking whether $[\![\phi]\!]\varnothing \cap \mathcal{M}_{\to} \,/\, \not\varnothing$.

2.3 State-Based Model Checking Abstractions

The classical state-based model checking can be formulated as an abstract interpretation, roughly abstracting traces to states.

Universal Checking Abstraction. Given a model (to check) $M \in \mathbb{M}$, the universal checking abstraction map $\alpha^{\forall}_M : \mathbb{M} \to \wp(\mathbb{S})$ abstracts a trace-interpreted temporal specification $\phi \in \mathbb{M}$ to the set of possible (present) states s of M which universally satisfy ϕ, that is, such that if the present state of M is s then ϕ holds. The intuition is that $\alpha^{\forall}_M(\phi)$ encodes a standard state-based interpretation like $\{s \in \mathbb{S} \mid M, s \models \phi\}$. The universal checking abstraction is therefore encoded by the following definition [11, Definition 45]:

$$\alpha^{\forall}_M(\phi) \stackrel{\text{def}}{=} \{s \in \mathbb{S} \mid M_{\downarrow s} \subseteq \phi\}.$$

Following the terminology by Müller-Olm et al. [16]: (i) the state-based global model checking problem of determining the set of present states in M that satisfy ϕ simply amounts to determining $\alpha^{\forall}_M(\phi)$, and (ii) the state-based local model checking problem of checking if a given state s in M satisfies ϕ amounts to checking whether $s \in \alpha^{\forall}_M(\phi)$.

In this case, the superset relation between states provides the notion of approximation. Actually, $\alpha^{\forall}_M$ gives rise to an adjunction between $\langle \mathbb{M}, \supseteq\rangle$ and $\langle \wp(\mathbb{S}), \supseteq\rangle$, and, together with its adjoint $\gamma^{\forall}_M$, induces the following closure operator on models.

Definition 2.3. The *universal checking closure* relative to a model $M \in \mathbb{M}$ is $\rho^{\forall}_M \stackrel{\text{def}}{=} \gamma^{\forall}_M \circ \alpha^{\forall}_M \in \mathrm{uco}(\langle \mathbb{M}, \supseteq\rangle)$ defined by $\rho^{\forall}_M = \lambda X.\{\langle i, \sigma\rangle \in M \mid M_{\downarrow \sigma_i} \subseteq X\}$. □

The intuition is that $\rho^{\forall}_M(X)$ throws away from X all those traces $\langle i, \sigma\rangle$ either which are not in M — these traces "do not matter", since $\alpha^{\forall}_M(\neg M) = \varnothing$ — or which are in M but whose present state σ_i does not universally satisfy X.

Existential Checking Abstraction. Dually, the existential checking abstraction map $\alpha^{\exists}_M : \mathbb{M} \to \wp(\mathbb{S})$ abstracts a given trace-interpreted temporal specification $\phi \in \mathbb{M}$ to the set of possible (present) states s of the model M which existentially satisfy ϕ, that is, for which there exists at least a trace of M which satisfies ϕ and whose present state is s. This leads to the following definition [11, Definition 49]:

$$\alpha^{\exists}_M(\phi) \stackrel{\text{def}}{=} \{s \in \mathbb{S} \mid M_{\downarrow s} \cap \phi \ / \ \varnothing\}.$$

It can be roughly said that the existential checking abstraction is useful for checking so-called safety properties of reactive systems, i.e., "bad things do not happen during executions". In fact, the subset relation formalizes the notion of approximation: if $\alpha^{\exists}_M(\phi) \subseteq S$ then each $s \ / \ S$ is such that if M is in state s then ϕ surely does not hold, and therefore any $T \supseteq S$ has to be understood as less precise than S. It turns out that $\alpha^{\exists}_M$ gives rise to an adjunction between $\langle \mathbb{M}, \subseteq \rangle$ and $\langle \wp(\mathbb{S}), \subseteq \rangle$, and hence to the following closure ($\gamma^{\exists}_M$ is the adjoint map).

Definition 2.4. The *existential checking closure* relative to a model $M \in \mathbb{M}$ is $\rho^{\exists}_M \stackrel{\text{def}}{=} \gamma^{\exists}_M \circ \alpha^{\exists}_M \in \mathrm{uco}(\langle \mathbb{M}, \subseteq \rangle)$ defined by $\rho^{\exists}_M = \lambda X.\{\langle i, \sigma \rangle \in \mathbb{T} \mid \langle i, \sigma \rangle \in M \Rightarrow M_{\downarrow \sigma_i} \cap X \ / \ \varnothing\} = \lambda X.\{\langle i, \sigma \rangle \in M \mid M_{\downarrow \sigma_i} \cap X \ / \ \varnothing\} \cup \neg M$. □

Here, the intuition is that $\rho^{\exists}_M$ adds to X any trace which is not in M — these can be considered meaningless as far as the existential checking of M is concerned, since $\alpha^{\exists}_M(\neg M) = \varnothing$ — and any trace in M whose present state existentially satisfies X.

State-Based (Abstract) Semantics. Given a total transition system $\langle \mathbb{S}, \to \rangle$ and its associated model $\mathcal{M}_{\to}$, the classical state-based semantics on $\wp(\mathbb{S})$ of a temporal formula is calculationally designed as the abstract semantics induced by the model checking abstractions seen above. This is an instance of the standard abstract interpretation methodology (as recalled in general terms by Cousot and Cousot in [11, Section 8]): basically, this amounts to abstract any model transformer of Definition 2.2 by the corresponding best correct approximation induced by the checking abstraction. For example, the predecessor transformer $\oplus : \mathbb{M} \to \mathbb{M}$ is abstracted to $\alpha^{\forall}_{\mathcal{M}_{\to}} \circ \oplus \circ \gamma^{\forall}_{\mathcal{M}_{\to}} : \wp(\mathbb{S}) \to \wp(\mathbb{S})$.

The general scenario is as follows. $\mathbb{E}^s \stackrel{\text{def}}{=} \mathbb{X} \to \wp(\mathbb{S})$ is the set of state environments. The checking abstractions $\alpha^{\forall}_M$ and $\alpha^{\exists}_M$ are extended pointwise to environments: $\dot{\alpha}^{\forall}_M$, $\dot{\alpha}^{\exists}_M : \mathbb{E} \to \mathbb{E}^s$, where, e.g., $\dot{\alpha}^{\forall}_M(\xi) \stackrel{\text{def}}{=} \lambda X \in \mathbb{X}.\alpha^{\forall}_M(\xi(X))$. The process of abstraction then compositionally leads to the following abstract state-based semantics for the $\overset{\curvearrowright}{\mu}$-calculus: $[\![\cdot]\!]^{\forall}_{\to}$, $[\![\cdot]\!]^{\exists}_{\to} : \mathfrak{L}_{\overset{\curvearrowright}{\mu}} \to \mathbb{E}^s \to \wp(\mathbb{S})$. These are inductively defined as one expects, following the lines of Definition 2.2. Thus, $[\![\phi]\!]^{\forall}_{\to}$ and $[\![\phi]\!]^{\exists}_{\to}$ correspond to the classical state interpretations of a temporal formula ϕ.

Soundness of the abstract state-based semantics is by construction: for any $\phi \in \mathfrak{L}_{\overset{\curvearrowright}{\mu}}$ and $\xi \in \mathbb{E}$, $\alpha^{\forall}_{\mathcal{M}_{\to}}([\![\phi]\!]\xi) \supseteq [\![\phi]\!]^{\forall}_{\to}\dot{\alpha}^{\forall}_{\mathcal{M}_{\to}}(\xi)$ and $\alpha^{\exists}_{\mathcal{M}_{\to}}([\![\phi]\!]\xi) \subseteq [\![\phi]\!]^{\forall}_{\to}\dot{\alpha}^{\exists}_{\mathcal{M}_{\to}}(\xi)$.

In general, completeness does not hold, even when the set of states is finite, i.e., the containments above may well be strict (see the finite counterexample given in [11, Counterexample (60)]). This means, for example, that there

exist a closed formula $\phi \in \mathfrak{L}_{\overset{\curvearrowright}{\mu}}$ and a state $s \in \mathbb{S}$ such that trace-based and state-based model checking for ϕ in s are not equivalent: $\mathcal{M}_{\rightarrow}, s \models_{trace} \phi$ (viz., $(\mathcal{M}_{\rightarrow})_{\downarrow s} \subseteq [\![\phi]\!]\varnothing$), while $\mathcal{M}_{\rightarrow}, s \not\models_{state} \phi$ (viz., $s \not\in [\![\phi]\!]^{\forall}_{\rightarrow}\varnothing$). Intuitively, incompleteness states that in order to deal with temporal specifications of the $\overset{\curvearrowright}{\mu}$-calculus, model checking algorithms should handle sets of traces instead that sets of states, and clearly this is unfeasible.

Cousot and Cousot [11] identified the model transformers causing such incompleteness and provided some sufficient conditions ensuring completeness. The first incomplete transformer for the universal checking abstraction is the predecessor operator $\oplus$, as shown in [11, Section 11.2]. Disjunction, namely set union, is the second incomplete model transformer, as observed in [11, Section 11.6]. The aforementioned sufficient conditions allow to identify some meaningful complete fragments of the $\overset{\curvearrowright}{\mu}$-calculus. This is the case, for example, of the $\mu^{\forall}_{+}$-calculus considered in [11, Section 13], which is complete for the universal checking abstraction and subsumes the classical $\forall$CTL logic. Finally, the reversal model transformer $\curvearrowleft$ is also incomplete, as shown by the example given in [17], although this is not explicitly mentioned in [11]. Of course, a dual reasoning can be made for the existential checking abstraction: here, the incomplete model transformers are predecessor, conjunction and reversal.

3 Making State-Based Model Checking Complete for Traces

We have seen that the possibility of defining a complete abstract operation on a given abstract domain A depends on A only. This means that completeness is an abstract domain property and therefore opens the relevant question of making an abstract interpretation complete by minimally extending or, dually, restricting the underlying abstract domain. Following [13], given a concrete interpretation $f : C \rightarrow C$ and an abstract domain $A \in \mathcal{L}_C$, the *absolute complete shell*[2] (resp., *core*) of A for f, when it exists, is the most abstract (resp., concrete) domain $A^s \in \mathcal{L}_C$ (resp., $A^c \in \mathcal{L}_C$) which extends (resp., restricts), viz. is more (resp., less) precise than A, and is complete for f. In other words, the absolute complete shell, respectively core, of A characterizes the least amount of information to be added to, respectively removed from, A in order to get completeness, when this can be done. These completeness problems have been solved in a constructive way by Giacobazzi et al. in [13].

Theorem 3.1 ([13, Theorem 5.10]). *Let $F \subseteq C \rightarrow C$ and $A \in \mathcal{L}_C$. If F is a set of continuous (i.e., preserving lub's of directed subsets) functions then the absolute complete shell and core of A for F exist, and they can be characterized, respectively, as greatest and least fix-points.*

In the following, we study the problem of making state-based model checking complete with respect to the trace semantics of the $\overset{\curvearrowright}{\mu}$-calculus by minimally

[2] [13] also introduces the concepts of relative complete shell and core, and this explains the use of the adjective absolute.

reducing the abstract domain of states $\wp(\mathbb{S})$. We will make the following assumption.

Hypothesis 3.2. *For any universal and existential checking closure, respectively $\rho^{\forall}_M$ and $\rho^{\exists}_M$, the model $M \in \mathbb{M}$ is such that for any $s \in \mathbb{S}$, $|M_{\downarrow s}| > 1$.*

This means that for any state s, $M_{\downarrow s}$ contains at least two traces, i.e., for any trace in the model M there exists at least one more trace in M with the same present state. This hypothesis is satisfied by any model generated by a total transition system. In fact, let $\mathcal{M}_{\rightarrow}$ be generated by a total transition system $\langle \mathbb{S}, \rightarrow \rangle$. Thus, if $s \in \mathbb{S}$ then, by the totality of the transition relation, there exists at least a trace in $\mathcal{M}_{\rightarrow}$ with present state s, i.e., there exists $\langle i, \sigma \rangle \in (\mathcal{M}_{\rightarrow})_{\downarrow s}$. If $\sigma_{i+1} = s = \sigma_i$ then we simply consider the trace $\langle i+1, \sigma \rangle \in (\mathcal{M}_{\rightarrow})_{\downarrow s}$. Otherwise, $\sigma_{i+1} \not= \sigma_i$, and therefore it is enough to consider, for example, the shifted path $\sigma^{+1} \stackrel{\text{def}}{=} \lambda j.\, \sigma_{j+1}$, which is different from the path σ, and the trace $\langle i-1, \sigma^{+1} \rangle$, which belongs to $(\mathcal{M}_{\rightarrow})_{\downarrow s}$.

We have chosen to present our results for the case of the existential checking closure $\rho^{\exists}_M$. The existential closure $\rho^{\exists}_M$ is defined on the concrete domain $\langle \mathbb{M}, \subseteq \rangle$ of models ordered by the subset relation, and therefore this simplifies both the technical approach and the intuition.

3.1 Absolute Complete Core for the Predecessor

As recalled in Section 2.3, the predecessor model transformer of Definition 2.2 is a source of incompleteness. Since $\oplus$ is clearly continuous, by Theorem 3.1, we know that the absolute core of the existential checking closure for $\oplus$ exists. The following results characterize this core.

Definition 3.3. Given $n \in \mathbb{N}$, define $\eta^n_{\exists_M} \stackrel{\text{def}}{=} \oplus^n \circ \rho^{\exists}_M \circ \ominus^n$. □

Note that $\eta^n_{\exists_M} \in \mathrm{uco}(\langle \mathbb{M}, \subseteq \rangle)$. Monotonicity follows by composition, while $\eta^n_{\exists_M}$ is extensive because $Y \subseteq \eta^n_{\exists_M}(Y) \Leftrightarrow \ominus^n Y \subseteq \rho^{\exists}_M(\ominus^n(Y))$. Idempotence is as follows: $\eta^n_{\exists_M}(\eta^n_{\exists_M}(Y)) = \oplus^n(\rho^{\exists}_M(\ominus^n(\oplus^n(\rho^{\exists}_M(\ominus^n(Y)))))) = \oplus^n(\rho^{\exists}_M(\rho^{\exists}_M(\ominus^n(Y)))) = \oplus^n(\rho^{\exists}_M(\ominus^n(Y))) = \eta^n_{\exists_M}(Y)$.

Theorem 3.4. *The absolute complete core $C^{\oplus}_{\exists_M}$ of $\rho^{\exists}_M$ for $\oplus$ exists and it is characterized as follows:*
(1) *The set of fix-points of $C^{\oplus}_{\exists_M}$ is $\{Y \in \mathbb{M} \mid \forall k \in \mathbb{N}.\ \ominus^k Y = \rho^{\exists}_M(\ominus^k Y)\}$.*
(2) $C^{\oplus}_{\exists_M} = \sqcup_{n \in \mathbb{N}} \eta^n_{\exists_M}$.

The following result provides a useful characterization of the absolute complete core $C^{\oplus}_{\exists_M}$ based on the structure of the underlying transition system. We use the following notation: given a transition system $\langle \mathbb{S}, \rightarrow \rangle$ and states $r, s \in \mathbb{S}$, for any $k > 0$, $r \stackrel{k}{\rightarrow} s$ iff $r = r_0 \rightarrow r_1 \rightarrow r_2 \rightarrow \ldots \rightarrow r_k = s$, where $\{r_1, ..., r_{k-1}\} \subseteq \mathbb{S}$.

Lemma 3.5. $\rho^{\exists}_M = \{(\bigcup_{s \in S} M_{\downarrow s}) \cup \neg M \mid S \subseteq \mathbb{S}\}$.

Theorem 3.6. *Let $M = \mathcal{M}_{\rightarrow}$, for some total transition system $\langle \mathbb{S}, \rightarrow \rangle$. Then, for any $S \subseteq \mathbb{S}$, $(\bigcup_{s \in S} M_{\downarrow s}) \cup \neg M \ / \in C^{\oplus}_{\exists_M}$ iff there exist $k > 0$, $q \in S$, $r \in \mathbb{S} \smallsetminus S$ and $t \in \mathbb{S}$ such that $q \xrightarrow{k} t$ and $r \xrightarrow{k} t$.*

Hence, by Lemma 3.5, Theorem 3.6 characterizes exactly all the fix-points of the closure $\rho^{\exists}_M$ which must be removed in order to get the absolute complete core $C^{\oplus}_{\exists_M}$. As a consequence, observe that it always holds that $\mathbb{T}, \neg M \in C^{\oplus}_{\exists_M}$: in fact, by Theorem 3.6, $\mathbb{T} = (\bigcup_{s \in \mathbb{S}} M_{\downarrow s}) \cup \neg M$ and $\neg M = (\bigcup_{s \in \varnothing} M_{\downarrow s}) \cup \neg M$.

By exploiting the constructive results above, we are also able to characterize the structure of transition systems whose models induce an existential checking closure which is complete for the predecessor. These are the transition systems $\langle \mathbb{S}, \rightarrow \rangle$ for which the transition relation $\rightarrow$ is injective: the relation $\rightarrow$ is *injective* whenever $\forall r, s, t \in \mathbb{S}. (r \rightarrow t \ \& \ s \rightarrow t) \Rightarrow r = s$.

Theorem 3.7. *Let $M = \mathcal{M}_{\rightarrow}$, for some total transition system $\langle \mathbb{S}, \rightarrow \rangle$. Then, $\rho^{\exists}_M$ is complete for $\oplus$ if and only if $\rightarrow$ is injective.*

It is worth noting that injectivity means that each computation step is reversible, i.e. the reverse transition system $\langle \mathbb{S}, \leftarrow \rangle$ obtained by reversing the transition relation is deterministic. This is the case of *reversible computations*, i.e. computations whose output uniquely defines the input [2]. Let us also observe that if $s \in \mathbb{S}$ is a stalling state, i.e. such that $s \rightarrow s$, then the injectivity of the transition relation requires that $t \ / \not\rightarrow$ for any $t \ / \not\leftarrow$, i.e., s cannot be reached by any other state.

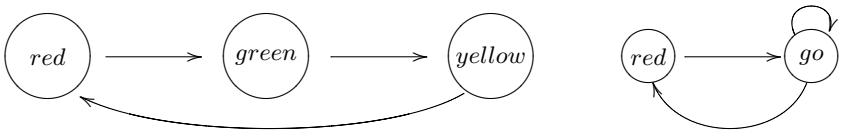


Fig. 1. A traffic light controller and its abstract version.

Example 3.8. Consider a traffic light controller as in Figure 1 with three states, i.e., $\mathbb{S} \stackrel{\text{def}}{=} \{red, green, yellow\}$ and $\rightarrow \ \stackrel{\text{def}}{=} \{red \rightarrow green, green \rightarrow yellow, yellow \rightarrow red\}$. Then, $\langle \mathbb{S}, \rightarrow \rangle$ is total and injective, and therefore, by Theorem 3.7, the corresponding existential checking closure is complete for the predecessor, i.e. $C^{\oplus}_{\exists_M} = \rho^{\exists}_M$.
Consider instead the abstraction induced by the following state partition: $h(red) = red$ and $h(green) = h(yellow) = go$ (cf. [6]). The resulting abstract transition system $\mathbb{S}^\sharp \stackrel{\text{def}}{=} \{red, go\}$ and $\rightarrow^\sharp \ \stackrel{\text{def}}{=} \{red \rightarrow^\sharp go, go \rightarrow^\sharp go, go \rightarrow^\sharp red\}$ is systematically derived from h as usual (cf. [6], see Figure 1). $\langle \mathbb{S}^\sharp, \rightarrow^\sharp \rangle$ is total but it is not injective. Let $M^\sharp$ be the model generated by $\langle \mathbb{S}^\sharp, \rightarrow^\sharp \rangle$. In order to compute the absolute complete core of $\rho^{\exists}_{M^\sharp}$ for $\oplus$ we exploit Theorem 3.6. Then, it is easy to verify that for any $\varnothing \ / \not\subseteq \subsetneq \mathbb{S}^\sharp$, i.e. either $S = \{go\}$ or $S = \{red\}$, the condition of Theorem 3.6 is satisfied. For example, for $S = \{go\}$, we have that

$red \rightarrow^{\sharp} go$ and $go \rightarrow^{\sharp} go$, with $go \in S$ and $red \not\in S$. Thus, the absolute complete core of $\rho^{\exists}_{M}$ for $\oplus$ is $C^{\oplus}_{\exists_M} = \{\mathbb{T}, \neg M\}$.
Actually, it is not difficult to show that any abstraction with at least two states of $\langle \mathbb{S}, \rightarrow \rangle$ induces an abstract transition system for which the existential closure is not complete for the predecessor. Of course, this is not always the case for abstract transition systems. In the case of an infinite counter modelled by a concrete transition system $\langle \mathbb{S}, \rightarrow \rangle$ where $\mathbb{S} = \mathbb{N}$ and $x \rightarrow y$ iff $y = x + 1$, it turns out that both $\langle \mathbb{S}, \rightarrow \rangle$ and the abstract transition system $\langle \{even,\ odd\}, \rightarrow^{p} \rangle$ with $\rightarrow^{p} \stackrel{\text{def}}{=} \{odd \rightarrow even, even \rightarrow odd\}$, obtained by the straightforward abstraction partitioning states in even and odd numbers, are such that the existential checking closure is complete for the predecessor: in fact, both transition relations are injective and therefore Theorem 3.7 can be applied. □

3.2 Absolute Complete Core for Time Reversal

Let us now analyze the time reversal operator $\curvearrowleft$. Also in this case $\curvearrowleft$ is trivially continuous on $\langle \mathbb{M}, \subseteq \rangle$, and therefore Theorem 3.1 guarantees the existence of the absolute core of the existential checking closure for $\curvearrowleft$. Then, let us characterize the existential checking closure for the reversed model.

Lemma 3.9. $\rho^{\exists}_{\curvearrowleft M} = \curvearrowleft \circ \rho^{\exists}_{M} \circ \curvearrowleft$.

The following characterization proves that the absolute complete core is given by those fix-points of $\rho^{\exists}_{M}$ which also belong to the existential checking closure $\rho^{\exists}_{\curvearrowleft M}$ relative to the reversed model $\curvearrowleft M$.

Theorem 3.10. *The absolute complete core $C^{\curvearrowleft}_{\exists_M}$ of $\rho^{\exists}_{M}$ for $\curvearrowleft$ exists and it is characterized as follows:*
(1) *The set of fix-points of $C^{\curvearrowleft}_{\exists_M}$ is $\{Y \in \mathbb{M} \mid Y, \curvearrowleft Y \in \rho^{\exists}_{M}\}$.*
(2) $C^{\curvearrowleft}_{\exists_M} = \rho^{\exists}_{M} \sqcup \rho^{\exists}_{\curvearrowleft M}$.

This result allows us to give a characterization of the structure of transition systems inducing an existential checking closure complete for time reversal. These are the transition systems with a symmetric transition relation ($\rightarrow$ is symmetric whenever $\forall r, s \in \mathbb{S}.\, r \rightarrow s \ \Rightarrow\ s \rightarrow r$), i.e. which allow both a computation and its reverse.

Corollary 3.11. *Let $M = \mathcal{M}_{\rightarrow}$, for some total transition system $\langle \mathbb{S}, \rightarrow \rangle$. Then, $\rho^{\exists}_{M}$ is complete for $\curvearrowleft$ if and only if $\rightarrow$ is symmetric.*

Thus, in practice, the existential checking closure is rarely complete for time reversal, since symmetry is not a realistic condition for most concrete transition systems.

Example 3.12. Consider the abstract transition systems of Example 3.8, namely the abstract counter and the abstract traffic light controller. For both systems, since the transition relations are symmetric, by Corollary 3.11, the existential checking closure is complete for time reversal. Instead, this is not the case for the concrete three-state traffic light controller, since the transition relation is not symmetric. □

3.3 Absolute Complete Core for Conjunction

Finally, let us consider conjunction, namely set intersection of models. Again, Theorem 3.1 ensures us that the absolute complete core does exist.

Theorem 3.13. *The absolute complete core* $C^{\cap}_{\exists_M}$ *of* $\rho^{\exists}_M$ *for* $\cap$ *exists and it is the top closure operator* $\lambda X.\mathbb{T}$.

Recall that the top closure operator corresponds to a straightforward and totally uninformative abstract domain consisting of a unique abstract value which is the abstraction of any concrete value. Hence, this is a "striking" result: in general, the top closure operator is always trivially complete (see [13]), and in our case this is the unique abstraction of $\rho^{\exists}_M$ which is complete for the conjunction. The intuition, that will be formally proved later on, is that any abstraction, but for the straightforward top closure, of the state-based model checking for a temporal calculus including an unrestricted connective of conjunction is incomplete for the trace-based semantics. This allows to state that state-based model checking is intrinsically incomplete for temporal calculi including conjunction, such as Kozen's μ-calculus, CTL, and CTL*.

3.4 Absolute Complete Core for All the Connectives

Finally, let us characterize the absolute complete core of $\rho^{\exists}_M$ for all the connectives of the $\overset{\curvearrowleft}{\mu}$-calculus, i.e., the set of all the model transformers of Definition 2.2. This core exists by Theorem 3.1 because all the operations are continuous. However, as noted by Ranzato [17], we must take care of the following technicality. As far as the universal state closure transformer $\forall$ is concerned, the following restriction is needed. We just consider the unary restrictions $\lambda X.\forall(N, X) : \mathbb{M} \to \mathbb{M}$, where $N \subseteq M \cup \curvearrowleft(M)$, of the universal state closure transformer, because the binary transformer $\forall : \mathbb{M} \times \mathbb{M} \to \mathbb{M}$ is neither monotone nor antitone in its first argument, and therefore it does not give rise to a concrete binary operation suitable to abstract interpretation. On the other hand, given any $N \in \mathbb{M}$, the unary restriction $\lambda X.\forall(N, X)$ is monotone. As observed in [11, Section 5] and [17, Section 3.1], this slight restriction still allows to recover the standard universal state quantification. In the sequel, we will use the following compact notation: $M^* \stackrel{\text{def}}{=} M \cup \curvearrowleft(M)$. As a simple consequence of Theorem 3.13, we get the following result.

Theorem 3.14. *The top closure operator* $\lambda X.\mathbb{T}$ *is the absolute complete core* $C_{\exists_M}$ *of* $\rho^{\exists}_M$ *for* $\{\boldsymbol{\sigma}_S\}_{S\in\wp(\mathbb{S})} \cup \{\boldsymbol{\pi}_t\}_{t\in\wp(\mathbb{S}^2)} \cup \{\oplus, \cap, \cup, \neg, \curvearrowleft\} \cup \{\lambda X.\forall(N, X)\}_{N\subseteq M^*}$.

The proof simply consists in observing that (1) the top closure $\lambda X.\mathbb{T}$ is trivially complete for any operation [13, Proposition 3.5 (i)], and (2) the absolute complete core of conjunction $\lambda X.\mathbb{T}$ is less than or equal to $C_{\exists_M}$, i.e., $C_{\exists_M} = \lambda X.\mathbb{T}$.

4 Completeness of Temporal Calculi

As observed above, from the abstract interpretation viewpoint, the universal state closure connective $\forall$ of the full $\overset{\curvearrowright}{\mu}$-calculus is somehow problematic, because, according to Cousot and Cousot's [11] Definition 2.1, the binary connective $\forall$ can be applied without any restriction, while its semantic counterpart, the universal state closure transformer $\forall : \mathbb{M} \times \mathbb{M} \to \mathbb{M}$, is neither monotone nor antitone in its first argument. On the other hand, given any $N \in \mathbb{M}$, the unary restriction $\lambda X.\forall(N, X) : \mathbb{M} \to \mathbb{M}$ is monotone, and this is enough in order to have the standard universal state quantification: $\forall\phi \stackrel{\text{def}}{=} \forall \boxplus (\boldsymbol{\pi}_\tau) : \phi$, where $[\![\boxplus (\boldsymbol{\pi}_\tau)]\!]\varnothing = \mathcal{M}_\tau$ (see [11, Section 5] for the details). Following Ranzato [17], this naturally leads to the following slight "monotone" restriction, which we call $\overset{\curvearrowright}{\mu}^-$-calculus, of the $\overset{\curvearrowright}{\mu}$-calculus.

Definition 4.1. Formulae ϕ of the $\overset{\curvearrowright}{\mu}^-$-calculus are inductively defined as follows:

$$\phi ::= \boldsymbol{\sigma}_S \mid \boldsymbol{\pi}_t \mid X \mid \oplus\, \phi \mid \phi^{\curvearrowleft} \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \boldsymbol{\mu} X.\phi \mid \boldsymbol{\nu} X.\phi \mid \forall\phi$$

where $S \in \wp(\mathbb{S})$, $t \in \wp(\mathbb{S} \times \mathbb{S})$, and $X \in \mathbb{X}$. $\mathfrak{L}_{\overset{\curvearrowright}{\mu}^-}$ denotes the set of $\overset{\curvearrowright}{\mu}^-$-calculus formulae. $\square$

Of course, the trace-semantics for the $\overset{\curvearrowright}{\mu}^-$-calculus is completely identical to that of the $\overset{\curvearrowright}{\mu}$-calculus given in Definition 2.2, but for the universal connective: $[\![\forall\phi]\!]\xi \stackrel{\text{def}}{=} \forall(\mathcal{M}_\tau, [\![\phi]\!]\xi)$.

The main result is stated for this $\overset{\curvearrowright}{\mu}^-$-calculus. The scenario is as follows. Any abstraction of the concrete domain $\mathbb{M}$ of temporal models, ordered by the superset or subset relation, induces an abstract semantics for the $\overset{\curvearrowright}{\mu}$-calculus, and therefore for the $\overset{\curvearrowright}{\mu}^-$-calculus. As seen in Section 2.3, the checking abstractions are an example: $[\![\cdot]\!]^{\forall}_{\to}$ and $[\![\cdot]\!]^{\exists}_{\to}$ are the abstract semantics induced, respectively, by $\rho^{\forall}_M$ and $\rho^{\exists}_M$. More in general, for the existential case of our interest, given a model to check $M \in \mathbb{M}$ — which is supposed to be generated by a transition system $\langle \mathbb{S}, \to \rangle$ — any closure operator, i.e. abstract domain, $\rho \in \mathrm{uco}(\mathbb{M}_\subseteq)$, induces the set of abstract environments $\mathbb{E}^\rho \stackrel{\text{def}}{=} \mathbb{X} \to \rho$, and the corresponding abstract semantics $[\![\cdot]\!]^\rho : \mathfrak{L}_{\overset{\curvearrowright}{\mu}^-} \to \mathbb{E}^\rho \to \rho$. Given an environment $\xi \in \mathbb{E}$, $\dot{\rho}(\xi) \stackrel{\text{def}}{=} \lambda X.\rho(\xi(X)) \in \mathbb{E}^\rho$ is the corresponding abstract environment induced by ρ. Soundness, i.e., $\forall\phi \in \mathfrak{L}_{\overset{\curvearrowright}{\mu}^-}.\forall\xi \in \mathbb{E}.\ \rho([\![\phi]\!]\xi) \supseteq [\![\phi]\!]^\rho \dot{\rho}(\xi)$, holds by construction (cf. [11, Theorem (40)]), while completeness for ρ means that equality always holds.

Theorem 4.2. *Let $\delta_\wedge$ be any fragment of the $\overset{\curvearrowright}{\mu}^-$-calculus which allows arbitrary conjunctions of formulae. The top closure operator $\lambda X.\mathbb{T} \in \mathrm{uco}(\mathbb{M}_\subseteq)$ is the greatest (w.r.t. subset image containment) closure operator on $\mathbb{M}_\subseteq$ (1) complete for $\delta_\wedge$ and (2) contained in the existential checking closure $\rho^{\exists}_M$.*

This result exactly formalizes the intuition described above: for any temporal language allowing unrestricted conjunctions of formulae, the top closure is the

unique abstraction of the existential state-based checking closure which induces a corresponding model checking which is complete for the trace-based semantics.

To conclude, let us mention that, dually, for the universal case, one gets the greatest closure operator on $\langle \mathbb{M}, \supseteq \rangle$, i.e. $\lambda X.\varnothing$.

5 Conclusion

This paper completed the study started by Ranzato [17] on the completeness of state-based model checking w.r.t. trace-based model checking. Both results show that abstract interpretation provides a powerful body of techniques to study the relation between computational models at different levels of abstraction. By using a slogan, this study showed that "*the state checking is intrinsically incomplete w.r.t. trace checking*", since no refinement or abstraction of the classical state-based model checking can lead to a semantics inducing a model checking which is complete for the trace semantics of the temporal language. This is not only a negative result concerning completeness of states vs. traces. This result opens new interesting research directions. In particular, in view of Theorem 3.7 and Corollary 3.11, it is possible to isolate fragments of μ-calculi which are complete for particular classes of transition systems. An important issue in this context is how completeness of state-based abstractions interacts with the presence of spurious counterexamples, when the transition systems are derived by abstract interpretation [1,6,14]. Completeness of states w.r.t. traces can also be studied from a different viewpoint. The idea is that of modifying the underlying temporal language in order to get completeness of states w.r.t. traces, where the modifications of the temporal language should be *minimal*. This means that temporal languages should be somehow ordered w.r.t. their expressive power, and this order should allow to define syntactically the minimal simplifications of the language. Further applications of our results could be obtained for temporal databases. Temporal logic has been proposed as the core language for specifying integrity constraints and triggers in temporal databases [3,18]. Since the time-point-indexed sequences (i.e. traces) of database states cannot be efficiently handled, several more efficient models have been obtained by abstraction. The problem of characterizing completeness in database abstractions has been attacked by several authors (e.g. see the notion of weak completeness in [8]), in particular for comparing different data-models. However, to the best of our knowledge, none of them have been considered as an abstract interpretation problem. In the case of temporal databases, temporal abstractions, such as the interval abstraction which induces the notion of period or temporal element [19], are particularly relevant. In view of recent results encoding temporal query languages as fragments of temporal logic [4], we believe that our results can help in comparing abstractions of temporal databases with the expressivity of temporal query languages, in particular for aspects concerning completeness with respect to a concrete temporal time-point-indexed database.

Acknowledgements. This work has been partially supported by the Italian MIUR Cofin2000 project "Abstract interpretation, type systems and control-flow analysis".

References

1. T. Ball, A. Podelski, and S.K. Rajamani. Relative Completeness of Abstraction Refinement for Software Model Checking. In *Proc. of TACAS'02*, LNCS 2280, pp. 158-172, Springer, 2002.
2. C.H. Bennett. Logical reversibility of computation. *IBM J. Research Dev.*, 21:905-940, 1981.
3. J. Chomicki. Temporal query languages: A survey. In *Proc. 1st Int. Conf. on Temporal Logic*, LNAI 827, pp. 506-534, Springer, 1994.
4. J. Chomicki, D. Toman, and M.H. Böhlen. Quering ATSQL databases with temporal logic. *ACM Trans. Database Syst.*, 26(2):145-1178, 2001.
5. E.M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 19(5):1512-1542, 1994.
6. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV'00*, LNCS 1855, pp. 154-169, Springer, 2000.
7. E.M. Clarke, O. Grumberg, and D. Peled. *Model checking*. The MIT Press, 1999.
8. J. Clifford, A. Croker, and A. Tuzhilin. On completeness of historical relational query languages. *ACM Trans. Database Syst.*, 19(1):64-116, 1994.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. ACM POPL'77*, pp. 238–252. ACM Press, 1977.
10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. ACM POPL'79*, pp. 269–282. ACM Press, 1979.
11. P. Cousot and R. Cousot. Temporal abstract interpretation. In *Proc. ACM POPL'00*, pp. 12–25. ACM Press, 2000.
12. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.
13. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000.
14. R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model checking. In *Proc. SAS'01*, LNCS 2126, pp. 356-373, Springer, 2001.
15. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods Syst. Des.*, 6:1–36, 1995.
16. M. Müller-Olm, D. Schmidt, and B. Steffen. Model checking: A tutorial introduction. In *Proc. SAS'99*, LNCS 1694, pp. 330–354. Springer, 1999.
17. F. Ranzato. On the completeness of model checking. In *Proc. ESOP'01*, LNCS 2028, pp. 137-154, Springer, 2001.
18. A.P. Sistla and O. Wolfson. Temporal triggers in active databases. *IEEE Trans. Knowl. Data. Eng.*, 7(3):471-486, 1995.
19. A.U. Tansel. Adding time dimension to relational model and extending relational algebra. *Information Systems*, 11(4):343-355, 1986.

Reuse of Results in Termination Analysis of Typed Logic Programs

Maurice Bruynooghe[1], Michael Codish[2], Samir Genaim[2]*, and Wim Vanhoof[1]

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

[2] Ben-Gurion University of the Negev, Department of Computer Science,
P.O.B. 653, 84105 Beer-Sheva, Israel
{mcodish,genaim}@cs.bgu.ac.il

Abstract. Recent works by the authors address the problem of automating the selection of a candidate norm for the purpose of termination analysis. These works illustrate a powerful technique in which a collection of simple type-based norms, one for each data type in the program, are combined together to provide the candidate norm. This paper extends these results by investigating type polymorphism. We show that by considering polymorphic types we reduce, without sacrificing precision, the number of type-based norms which should be combined to provide the candidate norm. Moreover, we show that when a generic polymorphic typed program component occurs in one or more specific type contexts, we need not reanalyze it. All of the information concerning its termination and its effect on the termination of other predicates in that context can be derived directly from the context independent analysis of that component based on norms derived from the polymorphic types.

1 Introduction

Termination analysis aims to determine that a given program definitely terminates for a specified, usually infinite, class of inputs. Proofs of termination are typically based on size functions (or *norms)* which map program states to the elements of a well founded domain. Termination is guaranteed if the states encountered through computation decrease in size.

For logic programs, loops occur through recursion and the size of a term is bounded if it is *rigid*. Namely, sufficiently instantiated so that its size does not change under further instantiation. Analyzers hence maintain two types of information: about the sizes of terms — to detect a decrease; and about their degree of instantiation — to detect rigidity. The ability to prove termination depends on finding a suitable norm and inferring sufficient size and instantiation information to detect with respect to that norm that all loops are decreasing and bound. In practice, most termination analyzers choose the natural numbers as the well-founded domain and measure size using semi-linear norms. Guessing a

* Supported by a PhD fellowship from the Kreitman Foundation.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 477–492, 2002.
© Springer-Verlag Berlin Heidelberg 2002

suitable norm reduces the level of intervention by the user and is often considered the main missing link in automatic termination analysis [11].

Termination analyzers typically rely on the user to select a candidate norm. Recently, some new ideas which are easily integrated into existing analyzers and facilitate automated norm selection emerged. In [26], a finite collection of norms is derived from the data types occurring in the program. The authors of [14] combine this collection in a single analysis and obtain a powerful system able to prove termination of programs not handled by previous implementations.

The current paper extends these ideas and shows that by considering polymorphic types we can reduce, without sacrificing precision, the number of norms which should be combined to provide the candidate norm. The intuition is simple: types which are instances of polymorphic typed variables need not be considered. All of the information required for termination analysis relevant to such types can be reconstructed from the polymorphic types. A preliminary investigation of the reuse of rigidity and size information based on polymorphic types are described in [6] and [26] respectively. However, these works focus mainly on monomorphic types and contain preliminary ideas for polymorphic types applicable only for simple instances of the polymorphic types.

The reuse of analysis results about polymorphic predicates is an important step in supporting modular termination analysis and in reducing the overall cost of termination analysis of large programs. It allows the analyzer to perform a polymorphic analysis of a module and to store results about the exported predicates in the module interface. The analysis of a module importing predicates in a specific type context can be completed by processing the results of the polymorphic analysis in the interface of the module defining the imported predicates (at least when there are no circular dependencies).

It has been recognized before that types provides a useful insight to the problem of guessing a norm [4,11,21,12,10] as recursive types represent recursive data-structures and thus identify potential sources of infinite recursion. The idea of combining norms has been suggested before by King *et al.* [16] in the context of lower-bound time-complexity analysis. For what concerns the problem of norm selection, [9] starts the analysis with a parameterized semi-linear norm and uses constraint solving to try to select one able to prove termination. Here we focus on (the combination of) type-based norms which are more refined than semi-linear norms and in particular on norms derived from polymorphic types.

The next section recalls some preliminaries about types. Section 3 reviews the previous work of the authors presented in [26] and [14] on defining type based norms and combining several norms in a single analysis. Section 4 presents the contribution of this paper extending the previous results for polymorphic types and proving that it is sufficient to consider norms corresponding to the polymorphic types without those for the types in the specific contexts in which they appear. We discuss related work in Section 5 and conclude in Section 6. A full version of this paper including detailed proofs and additional examples appears as [5].

2 Type Information

We assume familiarity with logic programming concepts [19,1]. We consider the set of terms constructed from a given set of function symbols Σ and variables V and use the notations f/n and p/n for an n-ary functor and predicate respectively. We adopt a standard notion of types for logic programs. Types, like terms are constructed from (type) variables and (type) symbols. We denote by $\mathcal{T} = \mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$ the set of types constructed from type variables $V_{\mathcal{T}}$ and type symbols $\Sigma_{\mathcal{T}}$. The sets of type variables, type symbols, variables and function symbols are assumed to be disjoint. Besides the usual substitutions which are mappings from variables to terms, we also have type substitutions which are mappings from type variables to types. For each type symbol, a unique type rule associates that symbol with a finite set of function symbols. We allow function symbols to be overloaded, i.e. they can be used in different type rules. A finite set of type rules forms a type definition and associates terms with types.

Definition 1 (type). *A* type rule *for a type symbol $h/n \in \Sigma_{\mathcal{T}}$ is a definition of the form $h(\bar{v}) \longrightarrow f_1(\bar{\tau}_1)\,;\ldots;\,f_k(\bar{\tau}_k)$ where: $\bar{v}$ is an n-tuple of distinct type variables from $V_{\mathcal{T}}$, $f_1, \ldots, f_k$ are distinct function symbols from Σ associated with the type symbol h, $\bar{\tau}_i$ $(1 \leq i \leq k)$ are corresponding tuples from $\mathcal{T}$, and type variables in $\bar{\tau}_i$, if any, are from $\bar{v}$. A* type definition *is a finite set of type rules for distinct type symbols. A type definition containing variables is said to be* polymorphic*, otherwise it is* monomorphic*.*

The application of a type substitution to a type rule gives an *instance* of the original rule. In what follows, we treat type rules as well as their instances as type rules.[1] In addition to types defined by the user, we allow also predefined types. An example is `int`.

Example 1. Below are several instances of the polymorphic type rule

```
type list(T) ---> [] ; [T | list(T)].
```

They define the monomorphic types $list(list(int))$ (for $T = list(int)$), $list(int)$ (for $T = int$) and the polymorphic type $list(list(V))$ (for $T = list(V)$).

```
type list(list(int)) ---> [] ; [list(int) | list(list(int))].
type list(int) ---> [] ; [int | list(int)].
type list(list(V)) ---> [] ; [list(V) | list(list(V))].
```
□

A monomorphic type definition determines the denotation of each type it defines and predefined types have a predefined denotation. For example, the terms 1 and 2 are of type int and $[1, 2]$ is of type $list(int)$. A polymorphic type definition is a schema that defines denotations for all its monomorphic instances. A common restriction in type based program analysis is that any monomorphic type implicitly defined by a polymorphic type definition can be defined by a finite set of monomorphic type rules that are instances of the polymorphic schema.

[1] Although in this case several rules for the same type symbol can occur.

This excludes type rules such as $t(T) \longrightarrow f(t(t(T)))$ and avoids termination problems of analysis.

The next definition specifies a notion of the *constituents* of a type. These are the possible types for subterms of terms of that type.

Definition 2 (type constituent). *Let τ be a type. We say that a type σ is a constituent of τ, denoted $\sigma \preceq \tau$, if there exists a term of type τ which has a subterm of type σ. The set of constituents of τ is denoted $Constituents(\tau)$. We denote by $Constituents(\rho)$ the union of the constituent sets for all types defined in the type definition ρ.*

Example 2. In the context of Example 1, observe that:

- $Constituents(T) = \{T\}$
- $Constituents(list(T)) = \{T, list(T)\}$
- $Constituents(int) = \{int\}$
- $Constituents(list(int)) = \{int, list(int)\}$
- $Constituents(list(list(V))) = \{V, list(V), list(list(V))\}$ □

In what follows, it is often convenient to consider the normal form of programs. In normal form, each atom is of the form $p(X_1, \ldots, X_n)$, $X = Y$ or $X = f(X_1, \ldots, X_n)$ (with $X_1, \ldots, X_n$) different variables). To develop our approach, we adopt the concept of *well-typing* of [23]. It relies on the notion of *variable typing*, which is a function from variables to types. First, it is assumed that each predicate p/n has a unique declaration $p(\tau_1, \ldots, \tau_n)$ stating the (possibly polymorphic) types of its arguments. Next, it is assumed that the program has a *well-typing*. This means that each clause $p(X_1, \ldots, X_n) \leftarrow B_1, \ldots, B_m \in P$ has a variable typing μ such that (1) $p(\mu(X_1), \ldots, \mu(X_n))$ is the declared type of p/n, (2) for each B_i: (a) If the atom is of the form $X = Y$, then $\mu(X) = \mu(Y)$. (b) If the atom is of the form $X = f(Y_1, \ldots, Y_l)$ then $\mu(X) = \tau$ such that τ is defined by a type rule (instance) including an alternative $f(\tau_1, \ldots, \tau_m)$ and $\mu(Y_i) = \tau_i$ for each i. (c) If the atom is of the form $q(Y_1, \ldots, Y_l)$ then $q(\mu(Y_1), \ldots, \mu(Y_l))$ is an instance of the type declared for q/l. The well-typing associates a type with each variable (and term) in the program. The notation $t : \tau$ is used to indicate that τ is the type of t.

Types may be declared by the user, as for example in Mercury [25], or inferred, as for example by the type inference system described in [13]. Requiring the existence of a well-typing is no limitation of our method. There is always a variable typing such that condition (1) is satisfied. Also, introducing fresh variables and extra unifications, one can always meet condition (2.c). Hence violations are limited to the unifications. Ignoring unifications that are not well-typed ensure that the analysis results are correct (although less precise than with a well-typing).

3 Termination Using Type Based Norms

We consider universal termination for well-typed logic programs using Prolog's leftmost selection rule and assume that unifications do not violate the occurs

check. Given a program P and a set of initial atomic queries S, we denote by $calls(P, S)$ the set of all atoms A such that a variant of A is a selected atom under the left-to-right selection rule in some derivation for a query $Q \in S$.

To measure the size of terms and atoms, one usually employs the notions of a *norm* and a *level mapping* which respectively map terms and atoms to natural numbers. Semi-linear norms [3] are well known. They map to the natural numbers and measure a term as the sum of the sizes of some of its arguments. For a term $f(t_1, \ldots, t_n)$, $\left|f(t_1, \ldots, t_n)\right| = c_f + \Sigma_{i \in I_f} \left|t_i\right|$ where constant c_f and indices $I_f \subseteq \{1, \ldots, n\}$ are determined by f/n; and for a variable X, $\left|X\right| = 0$.

A limitation of semi-linear norms is that the same function symbol f/n always makes the same contribution, c_f, independent of its context. Type based norms overcome this limitation. In our approach, a norm based on a type σ is defined to count the number of non-variable subterms of type σ in a term. It means that the contribution of a functor depends on the type context in which it appears. This simple idea turns out to be very powerful in practice and is captured in the following definition.

Definition 3 (type based norm). *Let ρ be a type definition and let σ be a type, predefined or defined in ρ. The σ-size of a term t of type τ, denoted $\left|t : \tau\right|_\sigma$, is computed by the following rules:*

1. *If $t = f(t_1, \ldots, t_n)$ is of type τ defined in ρ by the (unique) rule instance $\tau \longrightarrow f_1(\bar{\tau}_1)\,; \ldots\,;\, f_k(\bar{\tau}_k)$ and one of the $f_i(\bar{\tau}_i)$ on the right side of this rule is of the form $f(\tau_1, \ldots, \tau_n)$. Then, $\left|t : \tau\right|_\sigma = c(\sigma, \tau) + \left|t_1 : \tau_1\right|_\sigma + \cdots + \left|t_n : \tau_n\right|_\sigma$ where $c(\sigma, \tau) =$ if $(\sigma = \tau)$ then 1 else 0.*
2. *If t is a non-variable term of predefined type τ and $\tau = \sigma$ then $\left|t : \tau\right|_\sigma = 1$.*
3. *Otherwise $\left|t : \tau\right|_\sigma = 0$*

Note the role of the expression $c(\sigma, \tau)$ in the first rule of the definition. Recall that we are counting the number of subterms of type σ in a term $t = f(t_1, \ldots, t_n)$ of type τ. Hence, in addition to the contribution from $t_1, \ldots, t_n$, we should count t as such a subterm ($c(\sigma, \tau) = 1$) if $\sigma = \tau$, and otherwise not ($c(\sigma, \tau) = 0$). Observe also that the above definition is meaningful for polymorphic types, including a type variable T. The only terms of type T in a well-typed program are variables. Their T-size, as well as the T-size of any other term is 0. Similar definitions for monomorphic types can be found in [14,26].

Example 3. The following is the norm based on the type $list(int)$ from Example 1 as derived from Definition 3.

$$\left|t\right|_{list(int)} = \begin{cases} 1 + \left|t_2\right|_{list(int)} & \text{if } t = [t_1|t_2] : list(int) \\ 1 & \text{if } t = [\,] : list(int) \\ \left|t_1\right|_{list(int)} + \ldots + \left|t_n\right|_{list(int)} & \text{if } t = f(t_1, \ldots, t_n) : \tau \text{ and } \tau \not= list(int) \\ 0 & \text{otherwise} \end{cases}$$

Note that $\left|t\right|_{list(int)}$ is 0 when $list(int)$ is not a constituent of the type of t. Similar norms can be derived for int and $list(list(int))$ as well as for T and $list(T)$. As the only well-typed terms of type T are variables, the T-size of all well-typed terms is 0. The norm based on $list(T)$ is identical

to the one presented above after replacing the occurrences of "$list(int)$" by "$list(T)$". The following illustrates the σ-sizes for several terms t and types σ.

$t : list(list(int))$	$\|t\|_{int}$	$\|t\|_{list(int)}$	$\|t\|_{list(list(int))}$
$[\,]$	0	0	1
$[[\,]]$	0	1	2
$[[1]]$	1	2	2
$[[1,2],[3]]$	3	5	3

$t : list(T)$	$\|t\|_T$	$\|t\|_{list(T)}$
$[\,]$	0	1
$[X,Y]$	0	3

Note that equal subterms are measured differently according to the type context in which they appear. For instance, the typed term $[\,] : list(list(int))$ (first row on the left) measures 0 with $list(int)$-size, while $[\,] : list(int)$ (the inner $[\,]$ in the second row on the left), measures 1 with $list(int)$-size. □

In practice, a term can be measured only by a finite number of measures, one for each of the constituents of its type. That is because when σ is not a constituent of τ the σ-size of a term t of type τ is always zero. More generally, one obtains a finite set of candidate norms to be used in the analysis of a predicate, namely those determined by the constituents of the types in its definition.

For proving termination, it often suffices to consider constituents that define recursive data types, as termination typically depends on a decrease for some recursively defined data structure. However, for what concerns size and rigidity information of a polymorphic predicate, we should consider all polymorphic constituents, as these may occur in a context binding type variables to a recursive data type.

Rigidity generalizes groundness. A definition that applies on the constituents of polymorphically as well as monomorphically typed terms is as follows:

Definition 4 (type based rigidity). *With σ a constituent of τ, a term $t : \tau$ is σ-rigid iff for all variables $X : \tau'$ occurring in t, σ is not a constituent of τ'.*

Note that the σ-size of a σ-rigid term $t : \tau$ is constant for all instances $t\theta$ of type τ.

Our technique is built upon the semantic basis for termination analysis described in [7]. In this approach a program is mapped to a (in general, infinite) set of binary clauses which make answers, calls and loops observable in a goal independent way. A binary clause of the form $p \leftarrow q$ in the semantics indicates that a call to p leads to a subsequent call to q. A termination analyzer is obtained by approximating this semantics with respect to size and rigidity information. This gives a finite approximation of the calls and loops from which the analyzer can try to determine that the program terminates.

Rigidity can be represented in the domain Pos of positive Boolean functions [20] as shown in [15]. A type based rigidity analysis is developed in [6]. Size relations express linear information about the sizes of terms with respect to a given norm function.

Termination analysis for logic programs can be based (examples are in [7, 18,22]) on a technique termed abstract compilation. The program to be analyzed is first abstracted, using the chosen norm, to a corresponding constraint logic program which describes the size and rigidity dependencies specified by

the original program. As explained in [6], the rigidity abstraction is applied on the normal form. Unifications are abstracted by Pos-formulas modeling them. Program variables (in arguments of predicates and in unifications) are replaced by corresponding rigidity variables representing the rigidity of the term bound to the variable. The size abstraction can be obtained by systematically replacing terms by corresponding abstract terms. The abstraction of a term t with respect to the norm based on a type σ its called the *σ-abstraction* and is denoted by $|t|^{\alpha}_{\sigma}$. It is obtained by applying the norm to the argument, except that, whenever the norm is applied to a program variable it is mapped to a corresponding size variable representing its size. In addition, for each size variable X we add the constraint $X \geq 0$ (as size must always be non-negative). However, some refinements are possible. When abstracting X of type τ with respect to τ we add $X \geq 1$ as any τ-rigid instance must contain at least one term of type τ. Similarly, when abstracting X of type τ with respect to σ and τ does not have σ as constituent, then the constraint $X = 0$ can be added as well-typed instances of $X : \tau$ will never contain subterms of type σ. These refinements can be crucial for proving termination.

Note that the σ-abstraction of $t : \tau$ gives a size expression which generalizes the sizes of all instances of t. Namely, if t' is a term instance of t, then $|t'|_{\sigma}$ is a "size instance" of the abstraction of t. In particular, if t is a variable X then X_{σ} denotes the number of subterms of type σ in the particular instance of X.

The intuition is that where program variables range over terms, size variables range over the sizes of terms with respect to $|\cdot|$ and where a program predicate p/n represents an n-ary relation on terms, the corresponding abstract predicate represents an n-ary relation on the sizes of terms. The abstract program, P^a, abstracts the concrete program, P, in the sense that whenever $p(t_1, \ldots, t_n)$ is a consequence of P, then $p(|t_1|, \ldots, |t_n|)$ is a consequence of P^a.

When abstracting the binary clause semantics defined in [7] we derive finite sets of abstract binary clauses and abstract call patterns. The recursive abstract binary clauses describe the loops in the program and the call patterns describe the calls that arise in its computations. The key idea in the analysis is then that non-termination of the original program P with an initial query G implies "non-termination" in the abstract binary program for some call pattern with a single recursive abstract binary clause. In contrast to techniques based on acceptability [8], the binary clauses approach can apply different level mappings for proving acceptability of different binary clauses from the same predicate definition. This turns out to be an important factor when considering termination analysis based on a combination of norms as required in our approach.

Example 4. Consider the classic `append/3` program

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

occurring in two different type contexts as specified respectively by:

```
:- pred append(list(list(int)),list(list(int)),list(list(int))).
:- pred append(list(nest(int)),list(nest(int)),list(nest(int))).
```

with the type definitions from Example 1 and:

```
:- type list(nest(int)) ---> []; [nest(int)|list(nest(int))]
:- type nest ---> e(int) ; l(list(nest)).
```

The types declared in the second context allow to represent lists of lists (to any depth) of integers capturing all sublists by the same constituent, unlike definitions such as $list(\ldots(list(int)\ldots)$ where lists at different depths must be specified explicitly and are captured by different constituents.

For the purpose of termination analysis, we distinguish five type constituents for append in the given contexts: int, $list(int)$, $list(list(int))$, $nest(int)$ and $list(nest(int))$. The corresponding abstractions are given below.

The concrete term $[\,]$ is abstracted to 1 in the $list(int)$-, $list(list(int))$- and $list(nest(int))$-abstractions and to 0 in the others. The concrete term $[X|Xs]$ is abstracted respectively to $1+Xs$ in the $list(int)$- and $list(list(int))$-abstractions, to $X + Xs$ in the int- and $nest(int)$-abstractions, and to $1 + X + Xs$ in the $list(nest(int))$-abstraction.

list(list(int))- and list(int)-abstraction	int- and nest(int)-abstractions	list(nest(int))-abstraction

`append(1,Ys,Ys).`
`append(1+Xs,Ys,1+Zs):-`
`append(Xs,Ys,Zs).`
	`append(0,Ys,Ys).`
	`append(X+Xs,Ys,X+Zs):-`
	`append(Xs,Ys,Zs).`
		`append(1,Ys,Ys).`
		`append(1+X+Xs,Ys,1+X+Zs):-`
		`append(Xs,Ys,Zs).`

In either of these contexts, termination analysis using any one of these norms infers that all of the loops in this program component are of the form $append(x, y, z) \leftarrow append(u, v, w)$ with size information: $(u{<}x) \wedge (y{=}v) \wedge (w{<}z)$. Together with the respective instantiation information it infers that the component terminates for queries in which the first or third arguments are instantiated to rigid terms. There is a subtlety here: for the int-norm the proof of termination relies on the fact that the int-size of variable X is at least 1.

As all three abstractions lead to a proof of termination, one might conclude that it does not matter which single norm we consider. However, depending on the context in which append is called, we might need inter-argument size and instantiation information with respect to any or several of the different norms in order to prove the termination of other parts of the program. □

As an alternative to the approach described in Example 4, one can consider a polymorphic type declaration: `:- pred append(list(T),list(T),list(T))` which has only two constituents: T and $list(T)$. One can observe that the only terms in the program of type T are variables (namely X). The T-size of X is 0, however, its T-abstraction is X_T. The $list(T)$- and T- abstractions work out to be the same as the $list(int)$- (left) and int- (middle) abstractions in Example 4.

In the next section we show that these two abstractions contain all the information that we need to analyze the termination of append and other parts of the programs it might occur in. In our approach, we combine the abstractions corresponding to the polymorphic constituents (T and $list(T)$ in the example). Basically, we duplicate the arguments of each predicate, making fresh copies,

and apply one abstraction to each set of arguments. For example, in the program below copies of the variables are indexed by $list(T)$ and by T indicating the abstraction they represent.

combined abstraction

```
append(1,0,Ys_list(T),Ys_T,Ys_list(T),Ys_T).
append(1+Xs_list(T),X_T+Xs_T,Ys_list(T),Ys_T,1+Z_list(T),X_T+Zs_T):-
  append(Xs_list(T),Xs_T,Ys_list(T),Ys_T,Zs_list(T),Zs_T).
```

4 Polymorphic Reuse

Our aim is the following: We have a polymorphic typed program component P, for instance the append/3 relation typed with $list(T)$, and we have performed a type based termination analysis for P. Now we find predicates from P imported in a program Q imposing a specific type context. For instance, calls to append/3 in the contexts which manipulate lists of lists of integers and nested lists of integers as defined in Example 4). To analyze Q we want to reuse the analysis of P in its polymorphic context to: (1) consider the termination behavior of predicates defined in P for the more specific type context of Q, and (2) to reuse the size and rigidity information obtained for P based on polymorphic types to infer information based on the more specific type context in which it appears. This may be required for the termination analysis of the predicates defined in Q.

Type polymorphism is an important abstraction tool: a predicate defined with arguments of a polymorphic type can be called with actual arguments of any type that are instances of the defined types. We will argue two claims. First, that the polymorphic declarations are at least as precise as their instances as far as proving termination of a program. Namely, if we did not succeed to prove termination of a program using norms based on general polymorphic type definitions, then considering norms derived from more specific type information will not make the difference. Second, and more interesting, the size and rigidity information obtained during a termination analysis based on polymorphic type declarations for a program component can be reused to reason about termination of that component occurring in arbitrary more specific type contexts. Basing termination analysis on polymorphic types has another advantage. Recall that our proposal is to apply combined analysis considering each of the type constituents relevant to a predicate. Polymorphic type definitions are typically more conservative in the number of constituents involved and this reduces the number of norms combined and hence the cost of the analysis.

The key point to observe concerning the analysis of a program with polymorphic type declarations is that from the very essence of what a polymorphic type variable represents it follows that the program does not manipulate its "polymorphic parts". These are either completely ignored or simply passed from one data structure to another. Otherwise, if the program did manipulate a variable with a polymorphic type then that variable would have a more specific type. For

example, if a term `[X|Xs]` is declared to be of type `list(T)` then the type of X cannot be determined by the program. It is said to be "polymorphic".

The meaning of this observation as far as basing termination analysis on norms derived from polymorphic type information is that the polymorphic types contain already all the information relevant to termination also for any possible type instances. In particular, this means that the analysis based on norms for different instances of the polymorphic definition will always give the same result.

The basic question to answer is how do the size and rigidity abstractions with respect to a constituent of a type instance relate to those of the constituents of a polymorphic type. One could expect that there is a corresponding constituent in the polymorphic typing. However this is not always the case.

Let us assume a program component P typed by a polymorphic type definition ρ. The program may occur in several different type contexts each with more specific type information. For instance, append typed with $list(T)$ may occur in the contexts $list(int)$ or $list(color)$. Our goal is to reuse without loss of precision the termination analysis of P based on ρ when P occurs in different contexts. For the rest of this section let ρ' be a monomorphic type definition obtained by instantiating the type rules in ρ by a type substitution $\vartheta = \{T \mapsto \pi\}$ and adding rules to define any new types not already defined in ρ. The program (component) P is typed both by ρ as well as by ρ'.

The constituents of ρ' can be classified into two classes: $\mathcal{A}_{\rho'}$, those originating as instances of types defined in ρ; and $\mathcal{B}_{\rho'}$, those originating from the instantiation of ρ (constituents of π).

Example 5. Consider the following polymorphic type definition ρ and its instance ρ' obtained by the type substitution $\{T \mapsto color\}$ and the added rule to define $color$.

ρ (polymorphic)	ρ' (monomorphic)

`list(T) ---> [ ]; [T|list(T)].`	`list(color) --->`
	`[ ]; [color|list(color)].`
	`color --->`
	`red; blue; white; yellow.`

The monomorphic instance has two constituents giving $\mathcal{A}_{\rho'} = \{list(color)\}$ (because $list(color)$ corresponds to $list(T)$ defined in ρ), and $\mathcal{B}_{\rho'} = \{color\}$ (because the parameter T is mapped to $color$). In this case $\mathcal{A}_{\rho'}$ and $\mathcal{B}_{\rho'}$ are disjoint, however this is not always the case. Consider for example a definition ρ'' for type instance $list(nest(int))$ as defined in Example 4. One can observe that $list(nest(int))$ is a constituent of $nest(int)$ as well as an instance of $list(T)$. This implies that $list(nest(int)) \in \mathcal{A}_{\rho''} \cap \mathcal{B}_{\rho''}$. □

Let τ be a type defined in ρ and $\tau' = \tau[T \mapsto \pi]$ be its monomorphic instance defined in ρ'. Let t be a term of type τ as well as of type τ'. This means that the only subterms of type π in t are variables. We will show that the size (abstraction) of t with respect to any type σ' defined in ρ' can be described in terms of the

sizes (abstractions) of t with respect to some collection of types defined in ρ. The intuition is that counting subterms of type σ' in a term t is precisely like counting subterms of the types σ from which σ' originates. A similar intuition says that the σ'-rigidity of a term can be verified by checking the σ-rigidity of the term for the constituents σ corresponding to σ'. The following definition identifies the constituents in ρ corresponding to a constituent σ'.

Definition 5. *Let ρ be a polymorphic type definition and $\rho' = \rho[T \mapsto \pi]$ a monomorphic instance of ρ. We denote by $\mathcal{I}(\sigma')$ the type constituents in ρ that correspond to the constituent σ' of ρ'. It is defined as:*

$$\mathcal{I}(\sigma') = \begin{cases} \{\sigma \in \rho \mid \sigma' = \sigma[T \mapsto \pi]\} \cup \{T\} & \text{if } \sigma' \preceq \pi \\ \{\sigma \in \rho \mid \sigma' = \sigma[T \mapsto \pi]\} & \text{otherwise} \end{cases}$$

□

Example 6. Consider the type instances: $list(color)$, $list(int)$ and $list(nest(int))$ of $list(T)$. We have from Definition 5: $\mathcal{I}(color) = \mathcal{I}(int) = \mathcal{I}(nest(int)) = \{T\} \cup \{T\} = \{T\}$ from the first case in the definition ("$\sigma' \preceq \pi$"), $\mathcal{I}(list(color)) = \mathcal{I}(list(int)) = \{list(T)\}$ from the second case in the definition ("otherwise"), and $\mathcal{I}(list(nest(int)) = \{list(T)\} \cup \{T\} = \{T, list(T)\}$ also from the first case in the definition. □

We now proceed to state the main results of this paper. First we consider the termination of a program Q that calls predicates from a polymorphic typed, by ρ, program P, in a more specific type instance, ρ'. In order to prove the termination of Q we need to derive size and rigidity information of P with respect to the constituents of ρ'. The straightforward approach is to reanalyze P. The alternative contributed by this paper is to reconstruct this information from the analysis results obtained for P with the polymorphic constituents of ρ. The following theorem shows how to reconstruct this information.

Theorem 1. *Let p/n be a predicate typed under the polymorphic type definition ρ with k constituents: $\sigma_1, \ldots, \sigma_k$ and let:*

$$p(x_1^{\sigma_1}, \ldots x_1^{\sigma_k}, \ldots, x_n^{\sigma_1}, \ldots x_n^{\sigma_k}) \leftarrow \pi \in [\![P^\rho]\!]_{CLP(N)}$$
$$p(x_1^{\sigma_1}, \ldots x_1^{\sigma_k}, \ldots, x_n^{\sigma_1}, \ldots x_n^{\sigma_k}) \leftarrow \varphi \in [\![P^\rho]\!]_{Pos}$$

denote the size relation and rigidity information derived in the combined analysis using the type-based norms corresponding to these constituents (π is a linear constraint and φ is a Boolean formula). And Let:

$$p(x_1^{\sigma'_1}, \ldots x_1^{\sigma'_l}, \ldots, x_n^{\sigma'_1}, \ldots x_n^{\sigma'_l}) \leftarrow \pi' \in [\![P^{\rho'}]\!]_{CLP(N)}$$
$$p(x_1^{\sigma'_1}, \ldots x_1^{\sigma'_l}, \ldots, x_n^{\sigma'_1}, \ldots x_n^{\sigma'_l}) \leftarrow \varphi' \in [\![P^{\rho'}]\!]_{Pos}$$

denote the size relation and rigidity information with respect to a type definition ρ' which is an instance of ρ. Then:

$$\pi' \stackrel{def}{\equiv} \exists X : \pi \wedge \bigwedge_{j=1}^{l} \bigwedge_{i=1}^{n} (x_i^{\sigma'_j} = \sum_{\sigma \in \mathcal{I}(\sigma'_j)} x_i^{\sigma})$$
$$\varphi' \stackrel{def}{\equiv} \exists X : \varphi \wedge \bigwedge_{j=1}^{l} \bigwedge_{i=1}^{n} (x_i^{\sigma'_j} \leftrightarrow \bigwedge_{\sigma \in \mathcal{I}(\sigma'_j)} x_i^{\sigma})$$

Proof:. *A proof sketch is in [5].* □

In practice the analyzer uses widening while computing the size-relations; this can lead to differences in precision between the results obtained via reuse and the results obtained via direct computation. Only experience can show whether there is a relevant difference.

Example 7. Consider again the append/3 relation with the polymorphic type declaration ρ as given in Example 5. The size relation and rigidity information derived in the combined analysis using the type-based norms corresponding to the constituents T and $list(T)$ are:

$$\texttt{append}(\texttt{X}_T,\texttt{X}_{list(T)},\texttt{Y}_T,\texttt{Y}_{list(T)},\texttt{Z}_T,\texttt{Z}_{list(T)})\ \texttt{:-}$$
$$\texttt{X}_{list(T)}\texttt{+Y}_{list(T)}\texttt{=1+Z}_{list(T)}\ \wedge\ \texttt{X}_T\texttt{=Y}_T\texttt{+Z}_T$$
$$\texttt{append}(\texttt{X}_T,\texttt{X}_{list(T)},\texttt{Y}_T,\texttt{Y}_{list(T)},\texttt{Z}_T,\texttt{Z}_{list(T)})\ \texttt{:-}$$
$$((\texttt{X}_{list(T)}\ \wedge\ \texttt{Y}_{list(T)})\ \leftrightarrow\ \texttt{Z}_{list(T)})\ \wedge\ (\ \texttt{X}_T\ \leftrightarrow\ (\texttt{Y}_T\ \wedge\ \texttt{Z}_T))$$

we can derive the size relation and rigidity information for the combined analysis of $\sigma_1 = nest(int)$ and $\sigma_2 = list(nest(int))$ as follows:

$$\texttt{append}(\texttt{A}_{\sigma_1},\texttt{A}_{\sigma_2},\ \texttt{B}_{\sigma_1},\texttt{B}_{\sigma_2},\ \texttt{C}_{\sigma_1},\texttt{C}_{\sigma_2})\ \texttt{:-}$$
$$\texttt{A}_{\sigma_1}\ =\ \texttt{X}_T\ \wedge\ \texttt{A}_{\sigma_2}\ =\ \texttt{1+X}_T\texttt{+X}_{list(T)}\wedge$$
$$\texttt{B}_{\sigma_1}\ =\ \texttt{Y}_T\ \wedge\ \texttt{B}_{\sigma_2}\ =\ \texttt{Y}_{list(T)}\texttt{+Y}_T\ \wedge$$
$$\texttt{C}_{\sigma_1}\ =\ \texttt{Z}_T\ \wedge\ \texttt{C}_{\sigma_2}\ =\ \texttt{1+Z}_T\texttt{+Z}_{list(T)}\ \wedge$$
$$\texttt{X}_{list(T)}\texttt{+Y}_{list(T)}\texttt{=1+Z}_{list(T)}\ \wedge\ \texttt{X}_T\texttt{=Y}_T\texttt{+Z}_T$$
$$\texttt{append}(\texttt{A}_{\sigma_1},\texttt{A}_{\sigma_2},\ \texttt{B}_{\sigma_1},\texttt{B}_{\sigma_2},\ \texttt{C}_{\sigma_1},\texttt{C}_{\sigma_2})\ \texttt{:-}$$
$$\texttt{A}_{\sigma_1}\ \leftrightarrow\ \texttt{X}_T\ \wedge\ \texttt{A}_{\sigma_2}\ \leftrightarrow\ (\texttt{X}_{list(T)}\ \wedge\ \texttt{X}_T)\ \wedge$$
$$\texttt{B}_{\sigma_1}\ \leftrightarrow\ \texttt{Y}_T\ \wedge\ \texttt{B}_{\sigma_2}\ \leftrightarrow\ (\texttt{Y}_{list(T)}\wedge\ \texttt{Y}_T)\ \wedge$$
$$\texttt{C}_{\sigma_1}\ \leftrightarrow\ \texttt{Z}_T\ \wedge\ \texttt{C}_{\sigma_2}\ \leftrightarrow\ (\texttt{Z}_{list(T)}\wedge\ \texttt{Z}_T)\ \wedge$$
$$((\texttt{X}_{list(T)}\ \wedge\ \texttt{Y}_{list(T)})\ \leftrightarrow\ \texttt{Z}_{list(T)})\ \wedge\ (\texttt{X}_T\leftrightarrow\ (\texttt{Y}_T\ \wedge\ \texttt{Z}_T))$$

□

Another issue concerns the power of the analysis. Perhaps using type based constituents of a monomorphic instance, one can get an automated proof for a predicate that has no counterpart in the type based constituents of the polymorphic type. The next theorem shows this is impossible.

Theorem 2. *If termination of a predicate can be shown with type based norms derived from its monomorphic typing then termination can also be shown for type based norms derived from its polymorphic typing.*

Proof (intuition). If there is a proof of termination then there is a proof for every binary clause. Consider a binary clause (without loosing generality, we assume only one argument): `p(x) :-` π, `p(y)`. Firstly, if an argument position is rigid with respect to σ' of the monomorphic typing ρ' then, it is also rigid with respect to all $\sigma \in \mathcal{I}(\sigma')$ of the polymorphic typing ρ. A termination proof of the binary clause for the monomorphic typing implies that there exists a constituent σ' such that the σ'-size relation of π implies that $y^{\sigma'}$ is strictly less than $x^{\sigma'}$. But according to Theorem 1, the σ'-size relation of π is a linear combination of the σ-size relations of the constituents $\sigma \in \mathcal{I}(\sigma')$. Hence there must be at least

one constituent σ in the polymorphic typing such that the σ-size relation of π implies that y^σ is strictly less than x^σ. □

In fact, it appears that a polymorphic analysis could even be more powerful than a monomorphic one. If $\mathcal{I}(\sigma')$ is not a singleton, then it could be that the monomorphic σ' based norm shows no reduction for some binary clause while a polymorphic σ based norm for a constituent of $\mathcal{I}(\sigma')$ does. The reason is that what is counted by the σ' based norm can be split over several polymorphic norms and that one of these norms shows a decrease while their sum does not.

Example 8. Consider the following typed logic program:

```
:- type pair(T1,T2) ---> p(T1,T2).

:- pred q(pair(list(int),list(T)).
q(p([],_)).
q(p([X|Xs],Ys)) :-
     dupl(Ys,Ys,[],Zs),
     q(p(Xs,Zs)).

:- pred dupl(list(T),list(T),list(T),list(T)).
dupl([],Ys,Accm,Accm).
dupl([_|Xs],Ys,Accm,Zs) :-
     append(Ys,Accm,NewAccum),
     dupl(Xs,Ys,NewAccm,Zs).
```

Termination analysis based on the typed-norm $|.|_{int}$ obtain the binary clause $\mathtt{q(A)} \leftarrow [\mathtt{B{=}1{+}A}], \mathtt{q(B)}$ for $q/1$ which is sufficient for proving termination. But using the monomorphic instance which obtained by substituting T to int the analyzer fails to detect decreasing in the first argument of $q/1$ since the number subterm of type int is increasing. The same claim holds for $list(int)$.

□

All this shows that —ignoring the effects on precision caused by the widening of the size relations— it suffices for an automated type based termination proof to analyze a program component for the constituents of its polymorphic typing. Re-analyzing the program for the type based constituents of the context in which it appears is pointless.

5 Related Work

The idea of using type information to define norms for termination analysis has previously been studied by Bossi *et al.* [4], Martin *et al.* [21], and by Decorte *et al.* [11,12,10]. In this approach attention is focused on the class of programs, termination for which depends on disciplined manipulation of recursive data structures. Our approach to basing termination analysis on type information builds primarily on the technique of Decorte *et al.* and is described in [26] and [14]. These two works when combined together lead to an implemented system which avoids many of the difficulties and problems identified in the earlier works

in this area. The current paper extends this approach to deal with polymorphism and analysis reuse.

The use of types to refine other program analyses has recently been considered also in [6] and [17]. In those papers the authors observe that if a program is well-typed then only subterms of the same type can be unified. Hence, type information is used to refine the analysis of unifications in a program by considering for each type which subterms can be matched.

A preliminary exploration of polymorphism is in [26]. There, it observed that programs can be abstracted for polymorphic constituents. The authors notice that in the simple case where the instantiation $T \mapsto \pi$ of a polymorphic definition ρ does not interact with ρ itself then program abstraction for π information is identical to the polymorphic T-abstraction. The current paper handles the general case and shows how to reuse the polymorphic analyses in the context of other program components imposing more specific type contexts.

Modularity in termination analysis of logic programs has been considered before. See for example the recent paper [2] and references within. The motivation in these works is similar to ours. Namely by considering modules separately the required machinery or proof can be simplified (to focus on the individual module). In our approach this simplification stems from the fact that individual modules have more general (polymorphic) types. It is the polymorphic type definition for an individual module that captures the essence of how the module manipulates its (possibly more specific) data. For termination of that module we better focus on that more general information. An advantage of our approach is that it then spells out by looking at the instantiation of the polymorphic types in other modules exactly how to consider the combination and reuse of information between the modules.

6 Conclusion

This paper builds on the idea of basing termination analysis on type information. Recent work by the authors is presented in [26] and [14] and illustrates a powerful technique in which a collection of simple type-based norms — one for each data type in the program — are combined together to provide a candidate norm for termination analysis. An implementation based on these two works has proved to work well and avoid the difficulties described in earlier works.

The contribution of the current paper is to extend these previous results to work with polymorphic type information. Type polymorphism is an important abstraction tool and this turns out to be of practical importance when basing termination analysis on type information.

First because there are less types to consider when the definitions are more abstract; and second because even when a module occurs in one or more specific type contexts, all of the relevant information for that context can be obtained from the analysis based on its general, context independent, polymorphic types.

Acknowledgment. We thank John Gallagher and Vitaly Lagoon for the many helpful discussions on the issues presented in this paper.

References

1. K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.
2. A. Bossi, N. Cocco, S. Etalle and S. Rossi. On modular termination proofs of general logic programs. *Theory and Practice of Logic Programming*, 2(3):263–291, 2002.
3. A. Bossi, N. Cocco, and M. Fabris. Proving termination of logic programs by exploiting term properties. In S. Abramsky and T.S.E. Maibaum, editors, *Proceedings of Tapsoft 1991*, volume 494 of *Lecture Notes in Computer Science*, pages 153–180. Springer-Verlag, Berlin, 1991.
4. A. Bossi, N. Cocco, and M. Fabris. Typed norms. In B. Krieg-Brückner, editor, *Proceeedings ESOP '92*, volume 582 of *Lecture Notes in Computer Science*, pages 73–92. Springer-Verlag, Berlin, 1992.
5. M. Bruynooghe, M. Codish, S. Genaim, and W. Vanhoof. Reuse of results in termination analysis of typed logic programs. Technical Report, July, 2002.
6. M. Bruynooghe, W. Vanhoof, and M. Codish. Pos(t): Analyzing dependencies in typed logic programs. In *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, (PSI)*. Lecture Notes in Computer Science, vol. 2244. Springer-Verlag, 406–420.
7. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
8. D. De Schreye, C. Verschaetse, and M. Bruynooghe. A framework for analysing the termination of definite logic programs with respect to call patterns. In ICOT, editor, *Proc. of the Fifth Generation Computer Systems, FGCS92*, pages 481–488. ICOT, 1992.
9. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint based automatic termination analysis of Logic Programs. *ACM Trans. Program. Lang. Syst.*, 21(6):1137–1195, November 1999.
10. S. Decorte and D. De Schreye. Demand-driven and constraint-based automatic left-termination analysis for logic programs. In Naish [24], pages 78–92.
11. S. Decorte, D. De Schreye, and M. Fabris. Automatic inference of norms: A missing link in automatic termination analysis. In D. Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 420–436, Vancouver, Canada, 1993. MIT Press.
12. S. Decorte, D. De Schreye, and M. Fabris. Integrating types in termination analysis. Technical Report CW 222, K.U.Leuven, Department of Computer Science, January 1996.
13. J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002*, volume 2257 of *Lecture Notes in Computer Science*, pages 243–261. Springer, 2002.
14. S. Genaim, M. Codish, J. Gallagher, and V. Lagoon. Combining norms to prove termination. In Agostino Cortesi, editor, *Third International Workshop on Verification, Model Checking and Abstract Interpretation*, volume 2294 of *Lecture Notes in Computer Science*, pages 126–138. Springer-Verlag, January 2002.

15. R. Giacobazzi, S. K. Debray, and G. Levi. Generalized semantics and abstract interpretation for constraint logic programs. *The Journal of Logic Programming*, 25(3):191–247, Dec. 1995.
16. A. King, K. Shen, and F. Benoy. Lower-bound time-complexity analysis of logic programs. In Jan Maluszynski, editor, *International Symposium on Logic Programming*, pages 261 – 276. MIT Press, November 1997.
17. V. Lagoon and P. Stuckey. A framework for analysis of typed logic programs. In *FLOPS*, volume 2024 of *Lecture Notes in Computer Science*, pages 296–310. Springer-Verlag, Berlin, 2001.
18. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In Naish [24], pages 63–77.
19. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 2nd edition, 1987.
20. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.
21. J. Martin and A. King. Typed norms for typed logic programs. In *Logic Program Synthesis and Transformation*. Springer-Verlag, August 1996. Available at http://www.cs.ukc.ac.uk/pubs/1996/511.
22. F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logi programs. In *Static Analysis Symposium*, 2001.
23. A. Mycroft and R.A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence* 23(3): 295-307 1984.
24. L. Naish, editor. *Proceedings of Fourteenth International Conference on Logic Programming*, Leuven (Belgium), 1997. The MIT press.
25. 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, October–November 1996.
26. W. Vanhoof and M. Bruynooghe. When size does matter - Termination analysis for typed logic programs. *Logic Based Program Synthesis and Transformation, LOPSTR 2001, Revised Papers*, LNCS, Springer 2002. To appear.

Finiteness Analysis in Polynomial Time

Chin Soon Lee*

Datalogisk Institut, Copenhagen University, Denmark

Abstract. To achieve the termination of offline partial evaluation, it is necessary to ensure that static variables assume boundedly many values during specialization. Various works have addressed the analysis of variable boundedness, also called finiteness analysis, in the context of specializing first-order functional programs. The underlying reasoning is always: Unbounded sequences of increases in a static variable must be *impossible*, if they would give rise to unbounded sequences of size-decreases for some bounded-variable values.
Static analysis is used to collect a set of bipartite graphs that describe the parameter dependencies and data size changes in possible state transitions of the specializer (operating on the program). We capture the reasoning above as a condition on the graphs. This condition is decidable, but complete for PSPACE. We therefore derive a polynomial-time approximation, by considering realistic parameter size-change behaviour.

1 Introduction

1.1 Termination of Offline Partial Evaluation

Staging the computation. Partial evaluation is a *general* paradigm for program specialization. However, we will limit our discussion to the specialization of first-order functional programs.

Given a program and part of its input, a *residual program* is generated which, when given the remainder of the input, produces the same result as the original program given all of the input at once. The idea is to perform computations not depending on the missing part of the input, and thereby avoid these computations at runtime. This can be achieved by a combination of *unfolding*, *symbolic simplification* and *function specialization*.

Staging the partial evaluation. Offline partial evaluation implements a two-phase strategy [9]. First, binding-time analysis (BTA) is performed to classify variables as *static* or *dynamic*, to indicate whether a variable's values will be available during specialization. Such a classification is called a *binding-time division*. A division satisfies *congruence* if static variables are never assigned values depending on dynamic variables. Clearly, any absent input must be classified dynamic. Based on the binding-time division, each construct in the program is annotated static or dynamic, to indicate reduction or residualization.

* This research is supported by DAEDALUS, the RTD project IST-1999-20527 of the IST Programme within the European Union's Fifth Framework Programme.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 493–508, 2002.
© Springer-Verlag Berlin Heidelberg 2002

The next phase is fast, syntax-directed specialization: The binding-time of a subexpression's evaluation is *not* used to decide how to reduce an expression. This leads to a lightweight second phase.

Advantages over on-line methods, which make specialization decisions based on *type tags* for transformation-time values, include the following.

1. **Fast specialization**: Just as calling a specialized function or program multiple times increases the savings in execution cost, specializing an annotated function or program multiple times increases the savings in transformation cost, compared to specializing unannotated code.
2. **Annotations** can be consulted to understand the specializer's behaviour.
3. **Self application**: Specializing the specializer to an interpreter can yield an effective compiler [9]. This is an elegant approach to compiler generation, as only the interpreter has to be proven correct. Binding-time information is central to the success of self-application [5].

On the other hand, on-line partial evaluation can achieve more aggressive specialization. For example, conditional branches are not required to return values of matching binding-times, as required for syntax-directed specialization.

Two sources of non-termination. A congruent division is necessary to prevent binding-time type errors for offline partial evaluation, but it is not sufficient to guarantee a residual program.

There are two sources of non-termination for the specialization process. The first is infinite unfolding. This can be avoided by adapting methods for guaranteeing program termination. The principle: If every infinite unfolding would give rise to a sequence of static values whose sizes descend infinitely, then no infinite unfolding is possible [8,15,11]. For the rest of this paper, we will assume that finite unfolding has been ensured by marking sufficiently many function calls as dynamic. Refer to [3] for a thorough treatment of this issue.

Non-termination is also caused by accumulating parameters under dynamic control [9]. Consider specializing `f` below with `d` unknown and `s` equal to `[]`.

```
f(d,s) = if d=s then s else f(d,cons(1,s))
```

Suppose that the `f` call is annotated dynamic. Then the specializer would generate versions of `f` with `s` equal to `[]`, `[1]`, `[1,1]`, ... The standard prescription is to re-classify `s` as dynamic. This is known as *generalization*. We do not want to generalize *all* static computations controlled by a dynamic variable; the coding below, with `z` dynamic and `xs` static, is a useful way to coax the specializer to tabulate `h` with respect a finite set of values [10].

```
g(z,xs) = if z=hd(xs) then h(hd(xs)) else g(z,tl(xs))
```

These examples illustrate the concern of this article– a sufficiently precise analysis to determine that `xs` above is bounded during specialization, but point out that `s` is not bounded in the first example. Possibly unbounded variables are generalized. By repeatedly applying BTA, a safe unfolding strategy, and generalizations, the termination of specialization can be guaranteed.

1.2 Background of This Work

The analyses of variable boundedness in [7,4,1,3,8] all apply this reasoning: Unbounded sequences of *increases* in a static variable must be *impossible*, if they would give rise to unbounded sequences of size-decreases for some bounded-variable values. As such deductions rely on *known* bounded variables, it is necessary to formulate an explicit condition to collect such variables [4,3], or settle for a weak criterion to determine an initial set of them [7]. In any case, analysis of variable boundedness is complex. This led Neil Jones to adapt the above reasoning for a simpler but related problem. The *size-change principle* for program termination states that if an unbounded execution would give rise to unbounded size-decreases in a sequence of data values, then the program must terminate.

In joint work with Jones and Ben-Amram, a condition based on this principle was formulated and studied [14]. The size-change termination (SCT) condition is stated in terms of *size-change graphs*, bipartite graphs whose arcs describe decreasing and non-increasing data size changes in possible function-call transitions. A set of graphs satisfies SCT if every infinite flow-legal graph sequence (called a *multipath*) exhibits a connected sequence of arcs (called a *thread*) with infinitely many decreases. This condition is decidable, but complete for PSPACE.

In [13], we studied SCT approximations that work well in practice. They are based on trying to uncover a set of graphs $\mathcal{S}^\star$ that could be the *infinity set* of a multipath *without* infinite descent. Consider a maximal strongly-connected set of size-change graphs $\mathcal{S}$. We have developed ways to spot decreasing arcs that occur among a bounded number of threads in any $\mathcal{S}$-multipath. The graphs containing these arcs clearly cannot appear in $\mathcal{S}^\star$, so they can be removed from consideration. If after repeatedly removing graphs, no strongly-connected set remains, SCT is satisfied.

The decreasing arcs signal "progress" for the computation, and we normally expect to find such arcs easily. We have an efficient means to spot them among *fan-in free* graphs, which occur commonly in practice. This leads to a worst-case quadratic-time approximation of SCT. A more complicated approximation handles general graphs. In our extensive experiments, we have found that the approximations are as effective as SCT in practice. They are also generally much faster, and come with scalability guarantees. In this article, we adapt the polynomial-time descent detection for the analysis of variable boundedness.

1.3 Structure of This Article

Section 2 introduces the subject language L, its specialization, and state transitions for the specializer. Section 3 discusses the use of *abstract state transitions* to capture the parameter dependencies and data size changes in possible state transitions. Section 4 formulates the boundedness principle in terms of the abstract transitions. We show that the resulting condition is decidable but complete for PSPACE. Section 5 presents a polynomial-time approximation of this condition. We then describe a procedure to collect bounded variables, and demonstrate the procedure on a running example. Section 6 contains some concluding remarks.

2 Programs and Their Specialization

2.1 The Subject Language L

The syntax of L. The subject language L is a minimal first-order functional language with eager evaluation. Programs in L are generated by the following grammar, where $c \in \mathit{Cst}$ is a *constant*, $x, x_i \in \mathit{Var}$ are *variable identifiers*, $f \in \mathit{Fun}$ is a *function identifier*, and $op \in \mathit{Op}$ is a primitive operator.

$$\begin{array}{ll} p \in \mathit{Pgm} & ::= d_1 \ldots d_m \\ d_i, d \in \mathit{Def} & ::= f(x_1, \ldots, x_n) = e^f \\ e, e_i, e^f \in \mathit{Exp} & ::= c \mid x \mid op(e_1, \ldots, e_n) \mid \texttt{if}\ e_1\ e_2\ e_3 \mid f(e_1, \ldots, e_n) \end{array}$$

For any program, the sets *Def*, *Cst*, *Op*, *Var* and *Fun* are finite. We will assume that they are mutually exclusive.

The *definition* of f has the form $f(x_1, \ldots, x_n) = e^f$, where e^f is called the *body* of f. The number n of parameters is the function's *arity*, written $ar(f)$. Notation: $\mathit{Param}(f) = \{f^{(1)}, \ldots, f^{(n)}\}$ is the set of f parameters. In examples they are named by identifiers. Parameters are assumed to be in scope when they are used. Function calls are labelled, e.g., ${}^c f(e_1, \ldots, e_n)$. The set of function calls is denoted C. We assume that function calls and primitive operations have the correct number of arguments. Every program has a definition for the function `goal`. This function is never called in the program.

The Semantics of L. Programs in L are untyped. To provide L with a *big-step operational semantics* [6], we equip it with a syntactic category of *values* containing the distinguished element `true`. The set *Val* of values is a denumerable set. Its elements are usually written u, v. Operators are interpreted by a functional $\mathcal{O}$. For $op \in \mathit{Op}$, $\mathcal{O}[\![op]\!]$ is a *partial function* from Val^* (the set of finite value sequences) to *Val*. Constants are interpreted by a functional $\mathcal{K}$. For $c \in \mathit{Cst}$, $\mathcal{K}[\![c]\!] \in \mathit{Val}$. *Other notation*: For vector $\boldsymbol{v} = (v_1, \ldots, v_n)$ and $i = 1, \ldots, n$, let $(\boldsymbol{v})_i = v_i$.

Definition 1. *For $e \in \mathit{Exp}$, e evaluates to u on $\boldsymbol{v}$ if $\boldsymbol{v}, e \downarrow u$ is provable from the rules of Fig. 2. Otherwise, e is* undefined *on $\boldsymbol{v}$. Program p evaluates to u on $\boldsymbol{v}$ if* `goal` *evaluates to u on $\boldsymbol{v}$. Otherwise, p is* undefined *on $\boldsymbol{v}$.*

2.2 The 2-Level Language *2L*

In the first phase of offline partial evaluation, each construct in the subject program is annotated as static (to be reduced) or dynamic (to be made residual). Formally, this is a mapping from *Pgm* to *2Pgm*, a category of *2-level programs*.

$$\begin{array}{ll} tp \in \mathit{2Pgm} & ::= td_1 \ldots td_m \\ td_i, td \in \mathit{2Def} & ::= f(x_1, \ldots, x_n) = te^f \\ te, te_i, te^f \in \mathit{2Exp} & ::= c \mid x \mid \texttt{lift}(te) \mid op^{bt}(te_1, \ldots, te_n) \mid \\ & \qquad \texttt{if}^{bt}\ te_1\ te_2\ te_3 \mid f^{bt}(te_1, \ldots, te_n) \\ bt \in BT & ::= S \mid D \end{array}$$

As for L, te^f denotes the *body* of f in tp. The `lift`(te) construct directs the specializer to convert the result of evaluating te from a value to an expression. The set of values handled by the specializer is $2Val = Val \cup Exp$. We write a, b, a_i, b_i for $2Val$ values.

Definition 2. *For 2L program tp, $\phi(tp) \in Pgm$ is the program obtained by dropping all bt annotations and* `lift`*s. ϕ is called the* annotation-forgetting function.

Definition 3. *(Well annotations.)*

1. *A* BT environment $\tau : Var \cup Fun \to \{S, D\}$.
2. *The subject 2L program tp is* well-annotated *wrt τ if for each $f \in Fun$, $te^f : \tau(f)$ is provable from the rules of Fig. 3.*
3. τ respects binding-time assumptions *if $\tau(x) = S$ when x is a static input, $\tau(x) = D$ when x is a dynamic input and $\tau($`goal`$) = D$.*

For the remainder of the paper, we assume that we have been given tp and τ such that tp is well-annotated wrt τ. In general, given BT assumptions and L program p, it is possible to derive tp and τ such that τ respects the BT assumptions, tp is well-annotated wrt τ and $\phi(tp) = p$. We can also require certain variables and expressions to be dynamic. *Example*: The following is an interpreter for an L-like language, adapted from the one in [4]. The `case` construct is "syntactic sugar"; `body_of` and `param_of` are destructive operators; definitions for `lkvar`, `lkbody` and `lkpar` have been omitted. The program is well-annotated with the following variables static: `p`, `e1`, `p1`, `ns1`, `es2`, `p2`, `ns2`.

```
goal(p, vs) =
  ¹eval^S(body_of^S(hd^S(p)), p, param_of^S(hd^S(p)), vs)

eval(e1, p1, ns1, vs1) =
  case^S e1 of
  ("var",v) : ²lkvar^S(v, ns1, vs1)
  ("if",e11,e12,e13) : if^D ³eval^D(e11, p1, ns1, vs1)
                            ⁴eval^D(e12, p1, ns1, vs1)
                            ⁵eval^D(e13, p1, ns1, vs1)
  ("op",op,es) : apply^D(lift(op), ⁶evals^S(es, p1, ns1, vs1))
  ("let",v,e11,e12) : ⁷eval^D(e12, p1, cons^S(v, ns1),
                        cons^D(⁸eval^D(e11, p1, ns1, vs1), vs1))
  ("call",f,es) : ⁹eval^D(¹⁰lkbody^S(f, p1), p1, ¹¹lkpar^S(f, p1),
                          ¹²evals^S(es, p1, ns1, vs1))
evals(es2, p2, ns2, vs2) =
  if^S eq^S(es2,nil) lift([])
      cons^D(¹³eval^D(hd^S(es2), p2, ns2, vs2),
             ¹⁴evals^S(tl^S(es2), p2, ns2, vs2))
```

Specializing the interpreter to an object program successfully eliminates syntactic dispatch and variable dereference [9].

Definition 4. *(Specialization of 2L programs.)*

1. *Let* $f^{(i_1)}, \ldots, f^{(i_s)}$ *and* $f^{(j_1)}, \ldots, f^{(j_d)}$ *be the static and dynamic parameters of* f *respectively, according to BT environment* τ*. For* $\boldsymbol{a} = (a_1, \ldots, a_{ar(f)})$*, let* $SPos_f(\boldsymbol{a}) = (a_{i_1}, \ldots, a_{i_s})$ *and* $DPos_f(\boldsymbol{a}) = (a_{j_1}, \ldots, a_{j_d})$*.*
2. $\boldsymbol{a}$ respects annotations of f *if* $SPos_f(\boldsymbol{a}) \in Val^*$ *and* $DPos_f(\boldsymbol{a}) \in Exp^*$*. Further, if for* $f^{(k)}$ *dynamic,* $(\boldsymbol{a})_k$ *is the variable* $f^{(k)}$*, then* $\boldsymbol{a}$ *is a* 2-level input to f*.*
3. *For* f *such that* $\tau(f) = D$*, the* specialization *of* f *wrt* $\boldsymbol{v}$ *is denoted* $f_{\boldsymbol{v}}$*. Its definition is* $f_{\boldsymbol{v}}\ \boldsymbol{x} = e$*, where for some 2-level input* $\boldsymbol{a}$ *to* f*,* $SPos_f(\boldsymbol{a}) = \boldsymbol{v}$*,* $DPos_f(\boldsymbol{a}) = \boldsymbol{x}$*, and* $\boldsymbol{a}, te^f \Downarrow e$ *is provable from the rules of Fig. 4.*
4. *A set of definitions is* closed *if every referenced function has a definition.*
5. *A* specialization of *2L*-program *tp wrt static input* $\boldsymbol{v}$ *is a minimal closed set of specializations containing a definition for* $\texttt{goal}_{\boldsymbol{v}}$*.*

There are well-known problems connected with careless call unfolding: duplication of code, duplication of computation and residual programs that terminate more than their subject programs. These problems are overcome by annotating enough calls as dynamic [9]. We will not be concerned with them here.

Definition 5. *(Transitions among specializer states.)*

1. *For* $\boldsymbol{a}$ *(resp.* $\boldsymbol{b}$*) respecting annotations of* f *(resp.* g*),* $(f, \boldsymbol{a}) \overset{S}{\Rightarrow} (g, \boldsymbol{b})$ *if* $g^S(te_1, \ldots, te_n)$ *occurs in* te^f*, and for each* i*, we have* $\boldsymbol{a}, te_i \Downarrow (\boldsymbol{b})_i$*.*
2. *For* $\boldsymbol{a}$ *respecting annotations of* f *and* $\boldsymbol{b}$ *a 2-level input to* g*,* $(f, \boldsymbol{a}) \overset{D}{\Rightarrow} (g, \boldsymbol{b})$ *if* $g^D(te_1, \ldots, te_n)$ *occurs in* te^f*, and for* $\tau(g^{(i)}) = S$*, we have* $\boldsymbol{a}, te_i \Downarrow (\boldsymbol{b})_i$*.*
3. *Define* $\Rightarrow$ *to be* $\overset{S}{\Rightarrow} \cup \overset{D}{\Rightarrow}$*. Write* $\Rightarrow^*$ *for the transitive closure of* $\Rightarrow$*.*
4. *2L program tp satisfies* local termination *if for* $\boldsymbol{a}$ *respecting the annotations of* f*, every sequence of* $\overset{S}{\Rightarrow}$ *transitions beginning with* $(f, \boldsymbol{a})$ *is finite.*
5. *2L program tp satisfies* global termination *if for* $\boldsymbol{a}$ *respecting the annotations of* f*, the set* $\{(g, \boldsymbol{b}) \mid (f, \boldsymbol{a}) \Rightarrow^* (g, \boldsymbol{b})\}$ *is finite.*

Proposition 1. *Suppose that tp satisfies local termination. Then for any* $\boldsymbol{a}$ *respecting the annotations of* f*, there exists* b *such that* $\boldsymbol{a}, te^f \Downarrow b$ *is provable.*

Proposition 2. *Suppose that tp satisfies local and global termination. Then for any static input* $\boldsymbol{v}$ *for tp, the specialization of tp wrt* $\boldsymbol{v}$ *exists.*

Local termination implies that any residual function definition is successfully generated (given sufficient resources), while global termination implies that the set of residual function definitions to be generated is finite. Local termination is ensured by the unfolding strategy. This article is concerned with the more subtle question of global termination.

Definition 6. *A variable* $g^{(i)}$ *is* bounded *if for any* $\boldsymbol{a}$ *respecting the annotations of* f*, the set* $\{(\boldsymbol{b})_i \mid (f, \boldsymbol{a}) \Rightarrow^* (g, \boldsymbol{b})\}$ *is finite.*

Lemma 1. *Given local termination, if every static variable is bounded, then global termination is satisfied.*

3 Abstract Transitions

Definition 7. *(Data size changes.)*

1. *We will* assume *a size function,* $|\bullet| : Val \to I\!N$.
2. *For te a subexpression of* te^f *such that* $te : S$ *is provable, describe te as* non-increasing *on a static* $f^{(i)}$ *if for every* $\boldsymbol{a}$ *respecting f annotations,* $\boldsymbol{a}, te \Downarrow b$ *where* $b \not\models \texttt{undef}$ *implies* $|(\boldsymbol{a})_i| \geq |b|$*; te is described as* decreasing *on* $f^{(i)}$ *if* $|(\boldsymbol{a})_i| > |b|$.

Typically, the size of a list is its number of constructors. We will assume that for x static, $\texttt{hd}(x)$ and $\texttt{tl}(x)$ are decreasing on x. The composition of decreasing operations is decreasing, thus $\texttt{hd}(\texttt{tl}(x))$ is decreasing on x. An expression consisting of just a static variable x is clearly non-increasing on x.

Definition 8. *(Parameter dependence.) Fix* $\preceq$ *to be any* downward-closed *preorder for Val [7]. Let te be a subexpression of* te^f *such that* $te : S$ *is provable.*

1. *A* dependence description *for te is a subset* Δ *of* $\{\overline{\uparrow}, \uparrow\} \times \{f^{(i)} \mid \tau(f^{(i)}) = S\}$. $\uparrow(x)$ *(resp.* $\overline{\uparrow}(x)$*) in* Δ *indicates a* constructive (resp. safe) dependence.
2. *Let* $\sigma : X \to Val$*, where* $X = \{x \mid \uparrow(x) \in \Delta\}$*. Call b a* fresh value with respect to σ, Δ *if for some 2-level input* $\boldsymbol{a}$ *to f:*
   - $(\boldsymbol{a})_i = \sigma(f^{(i)})$ *for each* $\uparrow(f^{(i)}) \in \Delta$,
   - $\boldsymbol{a}, te \Downarrow b$ *where* $b \not\models \texttt{undef}$*, and*
   - $b \not\preceq (\boldsymbol{a})_i$ *for each* $\overline{\uparrow}(f^{(i)}) \in \Delta$,
3. *If for every* σ*, the set of fresh values wrt* σ, Δ *is finite, then* Δ *is* safe for *te*.

A possibility for $\preceq$ is the homeomorphic embedding relation. The definition is admittedly difficult to use. However, if $\boldsymbol{a}, te \Downarrow b$ and $b \not\models \texttt{undef}$ implies that b is embedded in $(\boldsymbol{v})_i$, then $\{\overline{\uparrow}(f^{(i)})\}$ is certainly safe for te. For example, in the annotated interpreter, $\{\overline{\uparrow}(\texttt{p1})\}$ is safe for $\texttt{lkbody(f,p1)}$ and $\texttt{lkpar(f,p1)}$. The following is *always* safe for te: $\{\uparrow(x) \mid x$ occurs in $te, \tau(x) = S\}$.

Definition 9. *(Abstract transitions.)*

1. *An* abstract transition α *has the form* $f \xrightarrow{\Delta,\Gamma} g$ *where* $f, g \in Fun$ *are called the* source *and* target *functions, and* Δ *and* Γ *are (essentially) bipartite graphs.*
2. *An element of* Δ *has the form* $x \xrightarrow{\delta} y$*, where* $x \in \{z \in Param(f) \mid \tau(z) = S\}$, $y \in \{z \in Param(g) \mid \tau(z) = S\}$ *and* $\delta \in \{\uparrow, \overline{\uparrow}\}$.
3. *An element of* Γ *has the form* $x \xrightarrow{r} y$*, where* $x \in \{z \in Param(f) \mid \tau(z) = S\}$, $y \in \{z \in Param(g) \mid \tau(z) = S\}$ *and* $r \in \{\overline{\downarrow}, \downarrow\}$.

Definition 10. *(Safe abstract transitions.)*

1. *Let* ${}^c g(te_1, \ldots, te_n)$ *be a function call in* te^f*. Then* $f \xrightarrow{\Delta,\Gamma} g$ *is* safe for *c if*
   - $f^{(i)} \xrightarrow{\downarrow} g^{(j)} \in \Gamma$ *implies that* te_j *is decreasing on* $f^{(i)}$.

- $f^{(i)} \overset{\overline{\downarrow}}{\rightarrow} g^{(j)} \in \Gamma$ *implies that* te_j *is non-increasing on* $f^{(i)}$.
- $\{\delta(f^{(i)}) \mid f^{(i)} \overset{\delta}{\rightarrow} g^{(j)} \in \Delta\}$ *is a safe dependence description for* te_j.

2. *A* set of abstract transitions $\mathcal{T}$ safe for *tp has an element, denoted* α_c, *safe for each call* c.

Definition 11. *The* dependency graph *has arcs corresponding to the dependency arcs in every* $\alpha_c \in \mathcal{T}$. *Its arc set is* $\{x \overset{\delta}{\rightsquigarrow}_c y \mid x \overset{\delta}{\rightarrow} y \in \Delta, \alpha_c = f \overset{\Delta,\Gamma}{\longrightarrow} g\}$. *Write* $x \rightsquigarrow^+ y$ *if* x *and* y *are connected by a non-empty sequence of* $\mathcal{D}$ *arcs.*

It is easy to compile a safe set of abstract transitions based on simple analysis of call arguments. The set depicted in Fig. 1 has been derived this way. In general, proper size and dependency analysis is needed for better precision. Figure 1 also shows the dependency graph due to the depicted abstract transitions. Safe dependence labels and most callsites have been omitted.

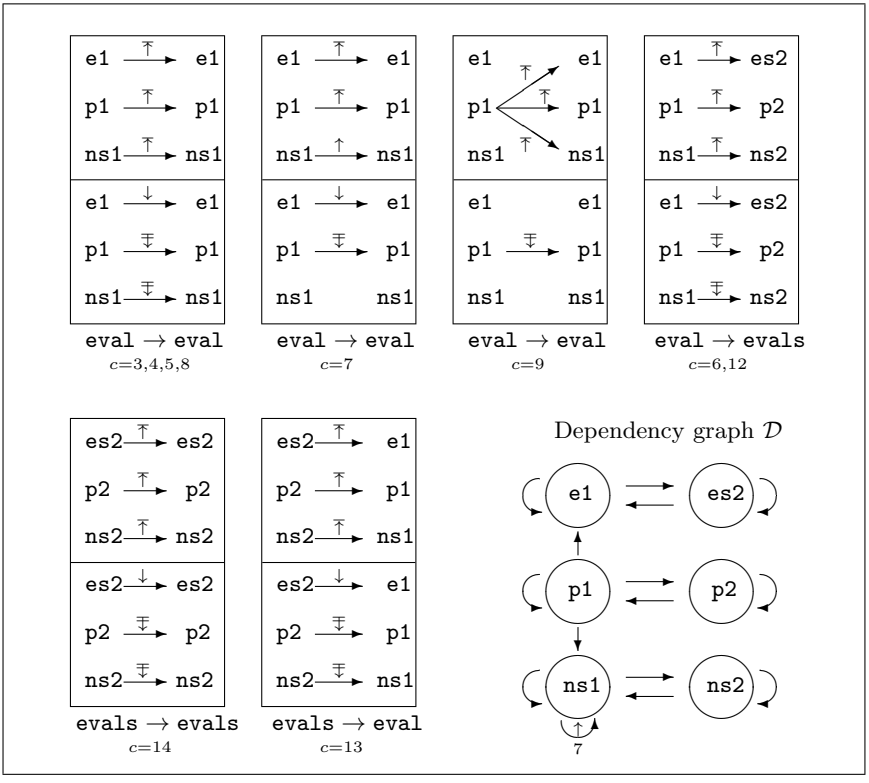


Fig. 1. Abstract transitions and dependency graph for the interpreter example

4 The Boundedness Principle

Flow-legal abstract transitions α_{c_1} and α_{c_2} can be composed in a natural way to summarize the parameter dependencies and data size changes observed in the call sequence c_1c_2.

Definition 12. *(Composition of abstract transitions)*

1. *For* $\alpha_{c_1} = f_0 \xrightarrow{\Delta,\Gamma} f_1$ *and* $\alpha_{c_2} = f_1 \xrightarrow{\Delta',\Gamma'} f_2$*, define* $\alpha_{c_1};\alpha_{c_2} = f_0 \xrightarrow{\Delta'',\Gamma''} f_2$*, where*

$$\begin{aligned}\Delta'' = \Delta;\Delta' &= \{x \xrightarrow{\uparrow} x'' \mid x \xrightarrow{\delta} x' \in \Delta, x' \xrightarrow{\delta'} x'' \in \Delta', \delta = \uparrow \text{ or } \delta' = \uparrow\} \cup \\ &\quad \{x \xrightarrow{\delta} x'' \mid x \xrightarrow{\delta} x' \in \Delta, x' \xrightarrow{\delta} x'' \in \Delta', \delta = \overline{\uparrow}\} \\ \Gamma'' = \Gamma;\Gamma' &= \{x \xrightarrow{\downarrow} x'' \mid x \xrightarrow{r} x' \in \Gamma, x' \xrightarrow{r'} x'' \in \Gamma', r = \downarrow \text{ or } r' = \downarrow\} \cup \\ &\quad \{x \xrightarrow{r} x'' \mid x \xrightarrow{r} x' \in \Gamma, x' \xrightarrow{r} x'' \in \Gamma', r = \overline{\downarrow}\}\end{aligned}$$

2. *The* composition closure *of abstract transitions* $\mathcal{T}$ *is the smallest* $\overline{\mathcal{T}}$ *satisfying* $\overline{\mathcal{T}} = \mathcal{T} \cup \{\alpha;\alpha' \mid \alpha, \alpha' \in \overline{\mathcal{T}}\}$.

Theorem 1. *We are given* z *with* $\tau(z) = S$*. Let* $X_0 = \{x \mid x \rightsquigarrow^+ z, z \not\rightsquigarrow^+ x\}$ *and* $X_1 = \{x \mid x \rightsquigarrow^+ z, z \rightsquigarrow^+ x\}$*. Then* z *is bounded if*

1. *for* $x \in X_0$*,* x *is bounded,* and
2. *for* $x \in X_1$*,* $\alpha = f \xrightarrow{\Delta,\Gamma} f \in \overline{\mathcal{T}}$ *such that* $\alpha;\alpha = \alpha$ *and* $x \xrightarrow{\uparrow} x \in \Delta$*, there exists some bounded* x' *for which* $x' \xrightarrow{\downarrow} x' \in \Gamma$*.*

Refer to [12] for a proof. Jones and Glenstrup explain the intuition behind Condition 2 in [8]. Intuitively, it expresses that unbounded sequences of construction (*increases*) on an X_1 variable would be accompanied by unbounded size-decreases in a bounded-variable's values. (It is a consequence of Ramsey's theorem that unbounded sequences of construction observed on an X_1 variable would entail sequences of state transitions described by unbounded repetitions of some α referred to in Condition 2.)

Theorem 2. *Condition 2 in Theorem 1 is complete for* PSPACE.

Proof. Given a set $\mathcal{G}$ of Γ graphs, it is PSPACE-hard to decide whether every Γ in the composition closure of $\mathcal{G}$ for which $\Gamma;\Gamma = \Gamma$ has an arc of the form $x' \xrightarrow{\downarrow} x'$ [14]. This problem is easily reduced to Condition 2 of Theorem 1. Let $\mathcal{T} = \{f \xrightarrow{\Delta,\Gamma} g \mid \Gamma \in \mathcal{G}, \Gamma$ arcs are from $Param(f)$ to $Param(g)$, $\Delta = \{x \xrightarrow{\uparrow} x\}\}$, where x is a fresh variable, every variable is deemed static, and every variable other than x is bounded. Then the condition holds for x just when every graph Γ in the composition closure of $\mathcal{G}$ for which $\Gamma;\Gamma = \Gamma$ has some arc $x' \xrightarrow{\downarrow} x'$.

The condition can clearly be refuted in non-deterministic polynomial-space. It can therefore be decided in PSPACE by standard techniques. □

5 A Polynomial-Time Boundedness Condition

We will adapt the approach in [13] (described briefly in Sect. 1.2) to approximate Condition 2 of Theorem 1. For simplicity, we will present a technique that works for *fan-in free* Γ graphs. Such graphs can be handled efficiently. They also arise in a natural way. A method in [13] can be adapted for general graphs.

Definition 13. *Γ is* fan-in free *if $x \xrightarrow{r} z, y \xrightarrow{r'} z \in \Gamma$ implies $x = y$.*

Henceforth, we assume that all Γ graphs are fan-in free.

Definition 14. *Let X' contain known bounded variables.*

1. *We write $G_c : f \to g$ and call G_c a* size-change graph *for c if $\alpha_c = f \xrightarrow{\Delta,\Gamma} g$ and G_c is the set of arcs $\{x \xrightarrow{r} y \in \Gamma \mid x, y \in X'\}$.*
2. *Define $\mathcal{G} = \{G_c \mid c \in C\}$.*

Definition 15. *For $\mathcal{G}' \subseteq \mathcal{G}$, the* descent-preservers wrt $\mathcal{G}'$, *denoted $DP(\mathcal{G}')$, is the largest subset of Var such that*

$$DP(\mathcal{G}') = \{x \mid \forall\, G \in \mathcal{G}', G : f \to g : \\ x \in Param(f) \Rightarrow \exists\, y \in DP(\mathcal{G}') : x \xrightarrow{r} y \in G \text{ for some } r\}$$

By design, any descent between $DP(\mathcal{G}')$ variables in a graph G of $\mathcal{G}'$ implies a descent between $DP(\mathcal{G}')$ variables in the composition of G with any $\mathcal{G}'$ graph.

Lemma 2. *Let $\mathcal{G}' \subseteq \mathcal{G}$. And let N and M be the space requirement for $\mathcal{G}$ and $\mathcal{G}'$ respectively. Then $DP(\mathcal{G}')$ can be computed in time $O(M)$, with an $O(N^2)$ initialization performed on $\mathcal{G}$.*

The algorithm maintains a set of counts for the source variables in each graph of $\mathcal{G}'$. These counts initially record the variable out-degrees. Start by marking every variable whose count is 0. Add these variables to a worklist. When processing x', inspect those graphs with arcs connected to x'. For such an arc $y' \xrightarrow{r} x'$, if y' is not yet marked, reduce its count. If the count becomes 0, mark y' and insert it into the worklist. Terminate when the worklist is empty. We claim, without proof, that the unmarked variables form $DP(\mathcal{G}')$ upon termination, and that the procedure has $O(M)$ time-complexity, with an $O(N^2)$ initialization performed on $\mathcal{G}$. (Initialization sets up the necessary indexing structures.)

Theorem 3. *As in Theorem 1, suppose that we are given z such that $\tau(z) = S$. And let $X_1 = \{x \mid x \rightsquigarrow^+ z, z \rightsquigarrow^+ x\}$. If Condition 2 of Theorem 1 is violated, there must exist a strongly-connected subgraph $\mathcal{D}^\star$ of $\mathcal{D}$ such that the following hold.*

- *$\mathcal{D}^\star$ has a vertex $x \in X_1$,*
- *$\mathcal{D}^\star$ has an arc of the form $x \overset{\uparrow}{\rightsquigarrow}_c y$,*

– *For* $C^\star = \{c \mid x \overset{\delta}{\rightsquigarrow}_c y \text{ in } \mathcal{D}^\star\}$ *and* $\mathcal{G}^\star = \{G_c \mid c \in C^\star\}$, *there do not exist* $x', y' \in DP(\mathcal{G}^\star)$ *and* $c \in C^\star$ *such that* $x' \overset{\downarrow}{\rightarrow} y' \in G_c$.

Proof. Suppose Condition 2 of Theorem 1 is violated, and let $\alpha = f \overset{\Delta,\Gamma}{\longrightarrow} f \in \overline{\mathcal{T}}$ be such that:

– $\alpha = \alpha; \alpha$,
– $x \overset{\uparrow}{\rightarrow} x \in \Delta$ for $x \in X_1$, and
– there does not exist any bounded z' such that $z' \overset{\downarrow}{\rightarrow} z' \in \Gamma$.

By definition of $\overline{\mathcal{T}}$, $\alpha = \alpha_{c_1}; \ldots; \alpha_{c_n} = f_0 \overset{\Delta_1,\Gamma_1}{\longrightarrow} f_1; \ldots; f_{n-1} \overset{\Delta_n,\Gamma_n}{\longrightarrow} f_n$, with $f_0 = f_n = f$. It is easily proved, by induction on n, that $\mathcal{D}$ has some arcs $x_0 \overset{\delta_1}{\rightsquigarrow}_{c_1} x_1, \ldots, x_{n-1} \overset{\delta_n}{\rightsquigarrow}_{c_n} x_n$ such that $x_0 = x_n = x$ and some $\delta_t = \uparrow$. Let $\mathcal{D}^\star$ be the subgraph of $\mathcal{D}$ comprised of these arcs, and vertices $x_0, \ldots, x_n$. Then $\mathcal{D}^\star$ is strongly-connected. It has a vertex $x \in X_1$ and an arc labelled $\uparrow$. Suppose that for $C^\star = \{c \mid x \overset{\delta}{\rightsquigarrow}_c y \text{ in } \mathcal{D}^\star\}$ and $\mathcal{G}^\star = \{G_c \mid c \in C^\star\}$, there are $x', y' \in DP(\mathcal{G}^\star)$ and $c \in C^\star$ such that $x' \overset{\downarrow}{\rightarrow} y' \in G_c$. We show that this leads to a contradiction.

For $i = 0, \ldots, n$, let $P_i = Param(f_i) \cap DP(\mathcal{G}^\star)$. By definition of DP and the assumption of no fan-ins, $|P_i| \leq |P_{i+1}|$ for $i = 0, \ldots, n-1$. Thus, we have that $|P_0| \leq |P_1| \leq \ldots \leq |P_n| = |P_0|$. Let $K = |P_0|$. It is easily proved, by induction on n, that $G = G_{c_1}; \ldots; G_{c_n}$ is fan-in free, and has exactly K arcs between DP-parameters, where one such arc is labelled $\downarrow$ (due to $x' \overset{\downarrow}{\rightarrow} y' \in G_c$). Moreover, $G; G = G$ implies that each arc of G between DP-parameters has the form $z' \overset{r}{\rightarrow} z'$. Therefore G has an arc of the form $z' \overset{\downarrow}{\rightarrow} z'$. By definition of size-change graphs, z' is bounded, so there exists a bounded z' such that $z' \overset{\downarrow}{\rightarrow} z' \in \Gamma$, which contradicts the premise. □

The idea then is to look for such a "problematic" $\mathcal{D}^\star$. We begin by considering maximal strongly-connected subgraphs of $\mathcal{D}$. Subgraphs *without* both a vertex $x \in X_1$ and a constructive arc need not be considered; their arcs can be removed from $\mathcal{D}$. Otherwise, if a subgraph has an arc $x \overset{\delta}{\rightsquigarrow}_c y$ such that G_c contains a decreasing arc between the DP-parameters, it is not hard to show that $x \overset{\delta}{\rightsquigarrow}_c y$ cannot be in any problematic $\mathcal{D}^\star$, so it can be removed from $\mathcal{D}$. If after repeatedly removing arcs from $\mathcal{D}$, no strongly-connected subgraph remains, then no problematic $\mathcal{D}^\star$ exists. It follows that Condition 2 of Theorem 1 holds.

5.1 Collecting Bounded Variables in Polynomial Time

A set of bounded variables can be collected as follows.

1. Compute the SCCs of $\mathcal{D}$, sorted in reverse topological order [2].
2. For each SCC X not marked as processed, mark it so, and call the procedure *Classify* below with the subgraph of $\mathcal{D}$ corresponding to X.

a) If *Classify* succeeds, mark all the variables of X as bounded.
b) Otherwise mark all the variables of X as possibly unbounded, and process every SCC reachable from X by marking their variables as possibly unbounded.

procedure *Classify*($\mathcal{D}'$)

1. If $\mathcal{D}'$ has no $\uparrow$ arc, succeed, else continue.
2. Compute $C' = \{c \mid x \overset{\delta}{\rightsquigarrow}_c y \text{ in } \mathcal{D}'\}$.
3. Compute $\mathcal{G}' = \{G_c \mid c \in C'\}$.
4. Compute $DP = DP(\mathcal{G}')$.
5. Compute $C^{\downarrow} = \{c \mid x', y' \in DP, x' \overset{\downarrow}{\rightarrow} y' \in G_c\}$.
6. If $C^{\downarrow} = \{\}$, fail, else continue.
7. Compute $\mathcal{D}''$, subgraph of $\mathcal{D}'$ excluding any arc $x \overset{\delta}{\rightsquigarrow}_c y$ where $c \in C^{\downarrow}$.
8. Call *Classify* on every maximal strongly-connected subgraph of $\mathcal{D}''$.

Lemma 3. *A variable classified as bounded* is *bounded.*

Proof. Consider a successful *Classify*($\mathcal{D}'$) call, where $\mathcal{D}'$ is a non-trivial strongly-connected component of $\mathcal{D}$. Let X_1 be the set of $\mathcal{D}'$ vertices. Then for any $z \in X_1$, we have $X_1 = \{x \mid x \rightsquigarrow^+ z, z \rightsquigarrow^+ x\}$. For $z \in X_1$, we claim that Condition 2 of Theorem 1 is satisfied. The lemma follows easily from this.

Suppose that Condition 2 of Theorem 1 does *not* hold. It follows from Theorem 3 that there exists a strongly-connected subgraph $\mathcal{D}^\star$ of $\mathcal{D}'$ such that $\mathcal{D}^\star$ has some arc of the form $x \overset{\uparrow}{\rightsquigarrow}_c y$, and for $C^\star = \{c \mid x \overset{\delta}{\rightsquigarrow}_c y \text{ in } \mathcal{D}^\star\}$ and $\mathcal{G}^\star = \{G_c \mid c \in C^\star\}$, there do not exist $x', y' \in DP(\mathcal{G}^\star)$ and $c \in C^\star$ such that $x' \overset{\downarrow}{\rightarrow} y' \in G_c$. We argue that this would cause *Classify*($\mathcal{D}'$) to fail, which contradicts our assumption.

Initially, the input to *Classify* includes $\mathcal{D}^\star$ as a subgraph. Now, if input $\mathcal{D}'$ to *Classify* includes $\mathcal{D}^\star$ and the call does not fail, then for $C' = \{c \mid x \overset{\delta}{\rightsquigarrow}_c y \text{ in } \mathcal{D}'\}$, $\mathcal{G}' = \{G_c \mid c \in C'\}$ and $C^{\downarrow} = \{c \mid x', y' \in DP(\mathcal{G}'), x' \overset{\downarrow}{\rightarrow} y' \in G_c\}$, we deduce that $C^{\downarrow}$ is non-empty, and that any $c \in C^{\downarrow}$ does *not* occur in $C^\star$ (this is because $DP(\mathcal{G}') \subseteq DP(\mathcal{G}^\star)$ for $\mathcal{G}^\star \subseteq \mathcal{G}'$). We conclude that $\mathcal{D}^\star$ is included in a maximal strongly-connected subgraph of the graph that remains, after removing from $\mathcal{D}'$ any arcs of the form $x \overset{\delta}{\rightarrow}_c y$ where $c \in C^{\downarrow}$, i.e., $\mathcal{D}^\star$ is included in the argument of a recursive call to *Classify*. Since each recursive call reduces the argument size, eventually some call has the argument $\mathcal{D}^\star$. This call must fail. □

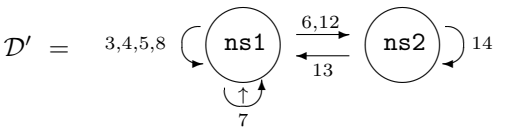
Lemma 4. *Bounded variables are collected in cubic time in the worst case.*

Proof. Both the dependency graph $\mathcal{D}$ and the size-change graphs are $O(N)$ in size, where N is the program size. For a call to *Classify* with argument $\mathcal{D}'$ of size M, the recursion depth is bounded by M. At each recursion level, we

calculate at most M DP sets, each taking time $O(N)$, and perform other linear-time operations on non-overlapping subproblems. Loosely, all the *Classify* calls at each recursion level takes time $O(MN)$. Thus calling *Classify* with $\mathcal{D}'$ has time complexity $O(M^2N)$. Finally, the procedure to collect bounded variables has time complexity of the order $\Sigma(M_i^2N)$, where $\Sigma M_i = N$. This is $O(N^3)$. □

The interpreter example. `goal` parameters are always bounded, and can be ignored. We begin with the dependency graph $\mathcal{D}$ in Fig. 1.

1. The SCCs of $\mathcal{D}$, in reverse topological order, are $\{\mathtt{p1}, \mathtt{p2}\}$, $\{\mathtt{e1}, \mathtt{es2}\}$ and $\{\mathtt{ns1}, \mathtt{ns2}\}$.
2. The subgraph of $\mathcal{D}$ corresponding to the component $\{\mathtt{p1}, \mathtt{p2}\}$ has no $\uparrow$ dependencies, so classify `p1` and `p2` as bounded.
3. Similarly, classify `e1` and `es2` as bounded.
4. The subgraph of $\mathcal{D}$ corresponding to the remaining component is depicted below. The corresponding set of calls is $C' = \{3, 4, 5, 6, 7, 8, 12, 13, 14\}$ (note the absence of 9). The set of G_c for $c \in C'$ has DP-set $\{\mathtt{e1}, \mathtt{es2}, \mathtt{p1}, \mathtt{p2}\}$, since each of these variables is connected to another one of the set in each G_c. Therefore, $C^{\downarrow} = C'$, since each G_c has one of the arcs: $\mathtt{e1} \xrightarrow{\downarrow} \mathtt{e1}$, $\mathtt{e1} \xrightarrow{\downarrow} \mathtt{es2}$, $\mathtt{es2} \xrightarrow{\downarrow} \mathtt{es2}$, or $\mathtt{es2} \xrightarrow{\downarrow} \mathtt{e1}$. The variables `ns1` and `ns2` are thus classified as bounded.



We see that all the static parameters are bounded for the interpreter example. Unbounded sequences of construction on `ns1` are impossible because such sequences would exhibit unbounded size-descent in the provably bounded `e1`, as detected by the procedure. The interest in this example is in call 9, which "resets" the value of `e1`. Fortunately, the interpreter implements lexical-scoping, thus the value of `ns1` is reset along with `e1`. For an interpreter implementing dynamic-scoping, the value of `ns1` would not be reset with `e1`, and our analysis would safely indicate that `ns1` may not be bounded.

6 Concluding Remarks

We have presented an approach to perform finiteness analysis in polynomial-time. As with such analyses in the literature [7,4,1,3], it is based on deducing that unbounded sequences of construction on a static variable during specialization would be accompanied by unbounded size decreases in some bounded-variable values, which is impossible. However, it focuses on realistic descent. Related

experiments [13] support this claim, and suggest that an analysis of variable boundedness based on Theorem 3 will be much faster than one based on Theorem 1, which captures the usual boundedness principle. Theorem 1 incorporates a condition that is *intrinsically* hard.

It is immediate future work to design and conduct experiments to test the effectiveness, efficiency and scalability of the proposed analysis.

Acknowledgements. Thanks go to Carl Frederiksen for his last-minute help with the paper.

References

1. Peter Holst Andersen and Carsten Kehler Holst. Termination analysis for offline partial evaluation of a higher-order functional language. In *Static Analysis, Proceedings of the Third International Symposium, SAS '96, Aachen, Germany, Sep 24–26, 1996*, volume 1145 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 1996.
2. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
3. Arne J. Glenstrup. Terminator II: Stopping partial evaluation of fully recursive programs. Master's thesis, DIKU, University of Copenhagen, Denmark, 1999.
4. Arne J. Glenstrup and Neil D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In *Perspectives of System Informatics, Proceedings of the Second International Andrei Ershov Memorial Conference, Russia, Jun 25–28, 1996*, volume 1181 of *Lecture Notes in Computer Science*, pages 273–284. Springer, 1996.
5. Robert Glück. Towards multiple self-application. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 309–320. ACM, 1991.
6. Matthew Hennessy. *The Semantics of Programming Languages*. Wiley, 1990.
7. Carsten Kehler Holst. Finiteness analysis. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, Aug 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 473–495. Springer, 1991.
8. Neil Jones and Arne Glenstrup. Program generation, termination, and binding-time analysis. In *Principles, Logics, and Implementations of High-Level Programming Languages, PLI 2002 (invited paper)*. Springer, 2002.
9. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
10. Chin Soon Lee. Partial evaluation of the euclidean algorithm, revisited. *Higher-Order and Symbolic Computation*, 12(2):203–212, Sep 1999.
11. Chin Soon Lee. *Program Termination Analysis, and Termination of Offline Partial Evaluation*. PhD thesis, UWA, University of Western Australia, Australia, 2001.
12. Chin Soon Lee. Finiteness analysis in polynomial time (with proofs). To be available as a technical report at DIKU, Copenhagen University, Denmark., 2002.
13. Chin Soon Lee. Program termination analysis in polynomial time. In *Proceedings of Generative Programming and Component Engineering, GPCE '02*. Springer, 2002.

14. Chin Soon Lee, Neil D. Jones, and Amir Ben-Amram. The size-change principle for program termination. In *Proceedings of the ACM Symposium on Principles of Programming Languages, Jan 2001*. ACM, 2001.
15. Wim Vanhoof and Maurice Bruynooghe. Binding-time annotations without binding-time analysis. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), 8th International Conference, Havana, Cuba, Dec 3-7, 2001*, volume 2250 of *Lecture Notes in Computer Science*, pages 707–722, 2001.

A Program and Specializer Semantics

$$\frac{u = \mathcal{K}[\![c]\!]}{\boldsymbol{v}, c \downarrow u} \qquad \frac{}{(v_1, \ldots, v_n), f^{(i)} \downarrow v_i}$$

$$\frac{\boldsymbol{v}, e_1 \downarrow \texttt{true} \quad \boldsymbol{v}, e_2 \downarrow u}{\boldsymbol{v}, \texttt{if}\ e_1\ e_2\ e_3 \downarrow u} \qquad \frac{\boldsymbol{v}, e_1 \downarrow u_1\ \ldots\ \boldsymbol{v}, e_n \downarrow u_n \quad u = \mathcal{O}[\![op]\!](u_1, \ldots, u_n)}{\boldsymbol{v}, op(e_1, \ldots, e_n) \downarrow u}$$

$$\frac{\boldsymbol{v}, e_1 \downarrow u_1 \quad u_1 \ / \ \cancel{\texttt{true}} \quad \boldsymbol{v}, e_3 \downarrow u}{\boldsymbol{v}, \texttt{if}\ e_1\ e_2\ e_3 \downarrow u} \qquad \frac{\boldsymbol{v}, e_1 \downarrow u_1\ \ldots\ \boldsymbol{v}, e_n \downarrow u_n \quad (u_1, \ldots, u_n), e^f \downarrow u}{\boldsymbol{v}, f(e_1, \ldots, e_n) \downarrow u}$$

Fig. 2. Evaluation of L expressions

$$\frac{}{c : S} \qquad \frac{}{x : \tau(x)}$$

$$\frac{te : S}{\texttt{lift}(te) : D} \qquad \frac{te_1 : bt\ \ldots\ te_n : bt}{op^{bt}(te_1, \ldots, te_n) : bt}$$

$$\frac{te_1 : S \quad te_2 : bt \quad te_3 : bt}{\texttt{if}^S\ te_1\ te_2\ te_3 : bt} \qquad \frac{te_1 : \tau(f^{(1)})\ \ldots\ te_n : \tau(f^{(n)})}{f^S(te_1, \ldots, te_n) : \tau(f)}$$

$$\frac{te_1 : D \quad te_2 : D \quad te_3 : D}{\texttt{if}^D\ te_1\ te_2\ te_3 : D} \qquad \frac{te_1 : \tau(f^{(1)})\ \ldots\ te_n : \tau(f^{(n)}) \quad \tau(f) = D}{f^D(te_1, \ldots, te_n) : D}$$

Fig. 3. Well-annotatedness of *2L* expressions

$$\frac{v = \mathcal{K}[\![c]\!]}{\boldsymbol{a}, c \Downarrow v} \qquad \frac{}{(a_1, \ldots, a_n), f^{(i)} \Downarrow a_i}$$

$$\frac{\boldsymbol{a}, te \Downarrow \texttt{undef}}{\boldsymbol{a}, \texttt{lift}(te) \Downarrow \texttt{error}()} \qquad \frac{\boldsymbol{a}, te \Downarrow v \quad \mathcal{K}[\![c]\!] = v}{\boldsymbol{a}, \texttt{lift}(te) \Downarrow c}$$

$$\frac{\boldsymbol{a}, te_1 \Downarrow \texttt{undef}}{\boldsymbol{a}, \texttt{if}^S \ te_1 \ te_2 \ te_3 \Downarrow \texttt{undef}}$$

$$\frac{\boldsymbol{a}, te_1 \Downarrow \texttt{true} \quad \boldsymbol{a}, te_2 \Downarrow b}{\boldsymbol{a}, \texttt{if}^S \ te_1 \ te_2 \ te_3 \Downarrow b} \qquad \frac{\boldsymbol{a}, te_1 \Downarrow v \quad v \notin \{\texttt{undef}, \texttt{true}\} \quad \boldsymbol{a}, te_3 \Downarrow b}{\boldsymbol{a}, \texttt{if}^S \ te_1 \ te_2 \ te_3 \Downarrow b}$$

$$\frac{\boldsymbol{a}, te_1 \Downarrow e_1 \quad \boldsymbol{a}, te_2 \Downarrow e_2 \quad \boldsymbol{a}, te_3 \Downarrow e_3 \quad e = \mathit{makeif}(e_1, e_2, e_3)}{\boldsymbol{a}, \texttt{if}^D \ te_1 \ te_2 \ te_3 \Downarrow e}$$

$$\frac{\boldsymbol{a}, te_1 \Downarrow v_1 \ \ldots \ \boldsymbol{a}, te_n \Downarrow v_n \quad (\exists\, i : v_i = \texttt{undef} \ \vee \ \nexists\, v : v = \mathcal{O}[\![op]\!](v_1, \ldots, v_n))}{\boldsymbol{a}, op^S(te_1, \ldots, te_n) \Downarrow \texttt{undef}}$$

$$\frac{\boldsymbol{a}, te_1 \Downarrow v_1 \ \ldots \ \boldsymbol{a}, te_n \Downarrow v_n \quad \forall\, i : v_i \ / \ \cancel{\texttt{undef}} \quad v = \mathcal{O}[\![op]\!](v_1, \ldots, v_n)}{\boldsymbol{a}, op^S(te_1, \ldots, te_n) \Downarrow v}$$

$$\frac{\boldsymbol{a}, te_1 \Downarrow e_1 \ \ldots \ \boldsymbol{a}, te_n \Downarrow e_n \quad e = \mathit{makeop}_{op}(e_1, \ldots, e_n)}{\boldsymbol{a}, op^D(te_1, \ldots, te_n) \Downarrow e}$$

$$\frac{\boldsymbol{a}, te_1 \Downarrow b_1 \ \ldots \ \boldsymbol{a}, te_n \Downarrow b_n \quad \exists\, i : b_i = \texttt{undef} \quad f^S(te_1, \ldots, te_n) : S}{\boldsymbol{a}, f^S(te_1, \ldots, te_n) \Downarrow \texttt{undef}}$$

$$\frac{\boldsymbol{a}, te_1 \Downarrow b_1 \ \ldots \ \boldsymbol{a}, te_n \Downarrow b_n \quad \exists\, i : b_i = \texttt{undef} \quad f^S(te_1, \ldots, te_n) : D}{\boldsymbol{a}, f^S(te_1, \ldots, te_n) \Downarrow \texttt{error}()}$$

$$\frac{\boldsymbol{a}, te_1 \Downarrow b_1 \ \ldots \ \boldsymbol{a}, te_n \Downarrow b_n \quad \forall\, i : b_i \ / \ \cancel{\texttt{undef}} \quad (b_1, \ldots, b_n), te^f \Downarrow b}{\boldsymbol{a}, f^S(te_1, \ldots, te_n) \Downarrow b}$$

$$\frac{\boldsymbol{a}, te_1 \Downarrow b_1 \ \ldots \ \boldsymbol{a}, te_n \Downarrow b_n \quad \exists\, i : b_i = \texttt{undef}}{\boldsymbol{a}, f^D(te_1, \ldots, te_n) \Downarrow \texttt{error}()}$$

$$\frac{\boldsymbol{a}, te_1 \Downarrow b_1 \ \ldots \ \boldsymbol{a}, te_n \Downarrow b_n \quad \forall\, i : b_i \ / \ \cancel{\texttt{undef}} \quad \boldsymbol{v} = \mathit{SPos}_f(\boldsymbol{b}) \quad \boldsymbol{e} = \mathit{DPos}_f(\boldsymbol{b})}{\boldsymbol{a}, f^D(te_1, \ldots, te_n) \Downarrow f_{\boldsymbol{v}} \ \boldsymbol{e}}$$

Remark: `undef` is a special value *not* in *Val*, and $\mathcal{O}[\![\texttt{error}]\!]()$ is left undefined.

Fig. 4. Specialization of *2L* expressions

Detecting Optimal Termination Conditions of Logic Programs

Fred Mesnard[1], Etienne Payet[1], and Ulrich Neumerkel[2]

[1] Iremia - Université de La Réunion, France
{fred, epayet}@univ-reunion.fr
[2] Institut für Computersprachen - T. U. Wien, Austria
ulrich@mips.complang.tuwien.ac.at

Abstract. In this paper, we begin with an approach to non-termination inference of logic programs. Our framework relies on an extension of the Lifting Theorem, where some specific argument positions can be instantiated while others are generalized. Atomic left looping queries are generated bottom-up from selected subsets of the binary unfoldings of the program of interest. Then non-termination inference is tailored to attempt proofs of optimality of left termination conditions computed by a termination inference tool. For each class of atomic queries not covered by a termination condition, the aim is to ensure the existence of one query from this class which leads to an infinite search tree. An experimental evaluation is reported. When termination and non-termination analysis produce complementary results for a logic procedure, they induce a *characterization* of the operational behavior of the logic procedure with respect to the left most selection rule and the language used to describe sets of atomic queries.

1 Introduction

Since the work of N. Lindenstrauss on TermiLog [16,10], several automatic tools for termination checking (e.g. TALP [4]) or termination inference (e.g. cTI [18, 19] or TerminWeb [13]) are now available to the logic programmer. One of them is even included in the Mercury compiler [24]. As the halting problem is undecidable for logic programs, such analyzers compute sufficient termination conditions implying left termination. In most works, only universal left termination is considered and termination conditions rely on a language for describing classes of atomic queries. The search tree associated to *any* (concrete) query satisfying a termination condition is guaranteed to be finite. When terms are abstracted using the *term-size* norm, then termination conditions are (disjunctions of) conjunctions of conditions of the form "the i-th argument is ground". Let us call this language $\mathcal{L}_{term}$.

In this report, we present the first approach to non-termination inference tailored to attempt proofs of *optimality* of termination conditions. The aim is to ensure the existence, for each class of atomic queries not covered by a termination condition, of *one* query from this class which leads to an infinite search tree. The main contributions of this work are:

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 509–525, 2002.
© Springer-Verlag Berlin Heidelberg 2002

- A generalization of the Lifting Theorem from the Logic Programming theory. The Lifting Theorem, at the heart of the completeness proof of SLD-resolution (see e.g. [3]), states that a SLD-derivation of $Q\theta$ can be lifted to a SLD-derivation ξ of Q. We prove that some specific arguments of Q, called "derivation neutral", can be instantiated as well, while retaining the existence of a lifted derivation ξ', where the length of ξ and ξ' are identical.
- A new application of binary unfoldings to left loop inference. [12] introduced the binary unfoldings of a logic program P as a goal independent technique to transform P into a possibly infinite set of binary clauses, which preserves the termination property [7] while abstracting the standard operational semantics associated to SLD-resolution. We present an algorithm to infer left looping classes of atomic goals, where such classes are computed bottom-up from selected subsets of the binary unfoldings of the analyzed program.
- An algorithm which, when combined with termination inference [17], may detect optimal left termination conditions expressed in $\mathcal{L}_{term}$ for logic programs.

We organize the paper as follows: Section 2 presents the notations. Then we define in Section 3 what we call an optimal termination condition. In Section 4 we propose an extension of the Lifting Theorem. We concentrate on non-termination inference in Section 5 and optimality proofs in Section 6.

2 Preliminaries

2.1 Logic Programming

We try to strictly adhere to the notations, definitions, and results presented in [1].

N denotes the set of non-negative integers and for any $n \in N$, $[1, n]$ denotes the set $\{1, \ldots, n\}$. If $n = 0$ then $[1, n] = \varnothing$.

Let $\mathcal{L}$ be a language of programs. We assume that $\mathcal{L}$ contains an infinite number of constant symbols including *void*. The set of relation symbols of $\mathcal{L}$ is Π, and we assume that each relation symbol p has an *unique* arity, denoted $arity(p)$. $TU_{\mathcal{L}}$ (resp. $TB_{\mathcal{L}}$) denotes the set of all (ground and non ground) terms of $\mathcal{L}$ (resp. atoms of $\mathcal{L}$). A *query* (or *goal*) $\mathbf{A}$ is a finite sequence of atoms $A_1, \ldots, A_n$ (where $n \geq 0$). Let t be a term. Then $Var(t)$ denotes the set of variables occurring in t.

A *logic program* is a finite set of definite clauses. In program examples, we use the ISO-Prolog syntax. Let P be a logic program. Then Π_P denotes the set of relation symbols appearing in P. In this paper, we only focus on universal left termination. Consider a non-empty query $A_1, A_2, \ldots, A_n$ and a clause c. Let $H \leftarrow \mathbf{B}$ be a variant of c variable disjoint[1] with $A_1, A_2, \ldots, A_n$ and assume that A_1 and H unify. Let θ be an mgu of A_1 and H. Then

[1] More generally, a variant c' of c satisfying the *standardization apart* condition: c' has to be variable disjoint from the initial query, the substitutions and the clauses used so far in the computation.

$A_1, A_2, \ldots, A_n \overset{\theta}{\underset{c}{\Longrightarrow}} (\mathbf{B}, A_2, \ldots, A_n)\theta$ is a *left derivation step* with $H \leftarrow \mathbf{B}$ as its *input clause*. If the substitution θ or the clause c is irrelevant, we drop a reference to it. We write $Q \overset{+}{\underset{P}{\Longrightarrow}} Q'$ (resp. $Q \overset{*}{\underset{P}{\Longrightarrow}} Q'$) to summarize a finite number (> 0) (resp. ≥ 0) of left derivation steps from Q to Q', where each input clause is a variant of a clause of P.

Let Q be a query. A *left derivation* of $\{Q\} \cup P$ is a *maximal* sequence of left derivation steps starting from the query Q, where each input clause is a variant of a clause of P. A finite left derivation may end up either with the empty query (then it is a *successful* left derivation) or with a non-empty query (then it is a *failed* left derivation). We say Q *left terminates* (resp. *left loops*) with respect to P if every left derivation of $\{Q\} \cup P$ is finite (resp. there exists an infinite left derivation of $\{Q\} \cup P$).

We recall that for logic programs, left termination is instantiation-closed: if Q left terminates with respect to P, then $Q\theta$ left terminates with respect to P' for any substitution θ and any $P' \subseteq P$. Similarly, left looping is generalization-closed: if there exists θ such that $Q\theta$ left loops with respect to P', then Q left loops with respect to any $P \supseteq P'$.

2.2 The Binary Unfoldings of a Logic Program

Let us present the main ideas about the *binary unfoldings* [12] of a logic program, borrowed from [7]. This technique transforms a logic program P (without any query of interest) into a possibly infinite set of binary clauses. Intuitively, each generated *binary clause* $H \leftarrow B$ (where B is either an atom or the atom *true* which denotes the empty query) specifies that, with respect to the original program P, a call to H (or any of its instances) necessary leads to a call to B (or its corresponding instance).

More precisely, let G be an atomic query. Then A is a *call* in a left derivation of $\{G\} \cup P$ if $G \overset{+}{\underset{P}{\Longrightarrow}} A, \mathbf{B}$. We denote by $calls_P(G)$ the set of calls which occur in the left derivations of $\{G\} \cup P$. The specialization of the goal independent semantics for call patterns for the left-to-right selection rule is given as the fixpoint of an operator T_P^β over the domain of binary clauses, viewed modulo renaming. In the definition below, *id* denotes the set of all binary clauses of the form $p(x_1, \ldots, x_n) \leftarrow p(x_1, \ldots, x_n)$ for any $p \in \Pi_P$, where $arity(p) = n$.

$$
\begin{aligned}
T_P^\beta(X) = \{(H \leftarrow B)\theta \mid\ & c := H \leftarrow B_1, \ldots, B_m \in P,\ i \in [1, m], \\
& \langle H_j \leftarrow true\rangle_{j=1}^{i-1} \in X \text{ renamed apart from } c, \\
& H_i \leftarrow B \in X \cup id \text{ renamed apart from } c, \\
& i < m \Rightarrow B \not\models true \\
& \theta = mgu(\langle B_1, \ldots, B_i\rangle, \langle H_1, \ldots, H_i\rangle)\}
\end{aligned}
$$

We define its powers as usual. It can be shown that the least fixpoint of this monotonic operator always exists and we set $bin\_unf(P) := lfp(T_P^\beta)$. Then the calls that occur in the left derivations of $\{G\} \cup P$ can be characterized as follows:

$calls_P(G) = \{B\theta | H \leftarrow B \in bin\_unf(P), \theta = mgu(G, H)\}$. This last property was one of the main initial motivations of the proposed abstract semantics, enabling logic programs optimizations. Similarly, $bin\_unf(P)$ gives a goal independent representation of the success patterns of P.

But we can extract more information from the binary unfoldings of a program P: universal left termination of an atomic goal G with respect to P is identical to universal termination of G with respect to $bin\_unf(P)$. Note that the selection rule is irrelevant for a binary program and an atomic query, as each subsequent query has at most one atom. The following result lies at the heart of Codish's approach to termination [7]:

Theorem 1 (Codish and Taboch, 99). *Let P be a program and G an atomic goal. Then G left loops with respect to P if and only if G loops with respect to $bin\_unf(P)$.*

Notice that $bin\_unf(P)$ is a possibly infinite set of binary clauses. For this reason, in the algorithms of sections 5 and 6, we compute only the first max iterations of T_P^β where max is a parameter of the analysis. As an immediate consequence of Theorem 1 frequently used in our proofs, assume that we detect that G loops with respect to a subset of the binary clauses of $T_P^\beta \uparrow i$, with $i \in N$. Then G loops with respect to $bin\_unf(P)$ hence G left loops with respect to P.

Example 1. Consider the following program P:

```
p(X,Z) :- p(Y,Z),q(X,Y).     p(X,X).     q(a,b).
```

The binary unfoldings of P are:

$$
\begin{aligned}
&T_{P'}^\beta \uparrow 0 = \varnothing \\
&T_{P'}^\beta \uparrow 1 = \{p(x,z) \leftarrow p(y,z), p(x,x) \leftarrow true, q(a,b) \leftarrow true\} \cup T_{P'}^\beta \uparrow 0 \\
&T_{P'}^\beta \uparrow 2 = \{p(a,b) \leftarrow true, p(x,y) \leftarrow q(x,y), p(x,y) \leftarrow q(z,y)\} \cup T_{P'}^\beta \uparrow 1 \\
&T_{P'}^\beta \uparrow 3 = \{p(x,b) \leftarrow q(x,a), p(x,b) \leftarrow q(y,a)\} \cup T_{P'}^\beta \uparrow 2 \\
&T_{P'}^\beta \uparrow 4 = T_{P'}^\beta \uparrow 3 = bin\_unf(P')
\end{aligned}
$$

The mere existence of the clause $p(x,z) \leftarrow p(y,z) \in T_P^\beta \uparrow 1$ implies that $\{p(x,b)\} \cup \{p(x,z) \leftarrow p(y,z)\}$ loops. Hence $\{p(x,b)\} \cup P$ left loops.

3 Optimal Termination Conditions

Let P be a logic program and p be a relation symbol $\in \Pi_P$, with $arity(p) = n$. First, we describe the language $\mathcal{L}_{term}$ presented in Section 1 for abstracting sets of atomic queries:

Definition 1 (Mode). *A* mode *m_p for p is a subset of $[1, n]$, and denotes the following set of atomic goals: $[m_p] = \{p(t_1, \ldots, t_n) \in TB_{\mathcal{L}} | \ \forall i \in m_p \ Var(t_i) = \varnothing\}$. The set of all modes for p, i.e. $2^{[1,n]}$, is denoted $modes(p)$.*

Note that if $m_p = \varnothing$ then $[m_p] = \{p(t_1, \ldots, t_n) \in TB_{\mathcal{L}}\}$. Since a logic procedure may have multiple uses, we generalize:

Definition 2 (Multi-mode). *A* multi-mode *for p is a set of modes for p, and denotes the following set of atomic queries:* $[M_p] = \cup_{m \in M_p}[m]$.

Note that if $M_p = \varnothing$, then $[M_p] = \varnothing$. Now we can define what we mean by termination and looping condition:

Definition 3 (Terminating mode, termination condition). *A* terminating mode *m_p for p is a mode for p such that any query $\in [m_p]$ left terminates with respect to P. A* termination condition *TC_p for p is a set of terminating modes for p.*

Definition 4 (Looping mode, looping condition). *A* looping mode *m_p for p is a mode for p such that there exists a query $\in [m_p]$ which left loops with respect to P. A* looping condition *L_p for p is a set of looping modes for p.*

As left termination is instantiation-closed, any mode that is "below" (less general than) a terminating mode is also a terminating mode for p. Similarly, as left looping is generalization-closed, any mode that is "above" (more general than) a looping mode is also a looping mode for p. Let us be more precise:

Definition 5 (Less_general, more_general). *Let M_p be a multi-mode for the relation symbol p. We set:*

$$less\_general(M_p) = \{m \in modes(p) \mid \exists m' \in M_p \ [m] \subseteq [m']\}$$
$$more\_general(M_p) = \{m \in modes(p) \mid \exists m' \in M_p \ [m'] \subseteq [m]\}$$

We are now equipped to present a definition of optimality for termination conditions:

Definition 6 (Optimal termination condition). *An* optimal *termination condition TC_p for p is a termination condition for p such that there exists a looping condition L_p verifying:*

$$modes(p) = less\_general(TC_p) \cup more\_general(L_p)$$

Otherwise stated, given a termination condition TC_p, if each mode which is not less general than a mode of TC_p is a looping mode, then TC_p *characterizes* the operational behavior of p w.r.t. left termination and our language for defining sets of queries.

Example 2. Consider the program `APPEND`:

```
append([],Ys,Ys).                                  % C1
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).     % C2
```

A well-known termination condition is $TC_{append} = \{\{1\},\{3\}\}$. Indeed, any query of the form `append(`t`,Ys,Zs)` or `append(Xs,Ys,`t`)`, where t is a ground term (*i.e.* such that $Var(t) = \varnothing$), left terminates. We have:

$$less\_general(TC_{append}) = \{\{1\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$$

On the other hand, `append(Xs,[],Zs)` left loops. Hence $L_{append} = \{\{2\}\}$ is a looping condition and $more\_general(L_{append}) = \{\varnothing, \{2\}\}$. Since $modes(append) = less\_general(TC_{append}) \cup more\_general(L_{append})$, we conclude that the termination condition TC_{append} is optimal.

We have already presented a tool for inferring termination conditions in [19]. We now describe the concepts underlying the inference of looping modes.

4 Neutral Arguments for Left Derivation

A basic idea in the work we present lies in identifying arguments in clauses which we can disregard when unfolding a query. For instance, the second argument of the non-unit clause of `append` in Example 2 is such a candidate. Moreover, a very common programming technique called *accumulator passing* (see e.g., [20] p. 21–25), always produces such patterns.

We first give a technical tool to describe specific arguments inside a program and present a generalization of the relation "is an instance of". In Subsection 4.2, we formalize the concept of derivation neutrality. Subsection 4.3 gives the main result, in the form of a generalized Lifting Theorem, with an application to loop checking.

4.1 Sets of Positions

Definition 7 (Set of positions). *A* set of positions *is a mapping τ that maps each predicate symbol $p \in \Pi$ to a subset of* $[1, arity(p)]$.

Example 3 (Example 2 continued). If we want to disregard the second argument of the relation symbol *append*, we set $\tau := \langle append \mapsto \{\} \rangle$.

Definition 8 (τ-instance and τ-generalization). *Let τ be a set of positions. We make use of the following relations:*

- *The relation $=_\tau$:*

$$A =_\tau B \quad \textit{iff} \quad \begin{cases} A = p(s_1, \ldots, s_n) \\ B = p(t_1, \ldots, t_n) \\ \forall i \in [1, n] \setminus \tau(p),\ t_i = s_i \end{cases}$$

$$A_1, \ldots, A_n =_\tau B_1, \ldots, B_m \quad \textit{iff} \quad \begin{cases} n = m \\ \forall i \in [1, n],\ A_i =_\tau B_i \end{cases}$$

- *The relation "is a τ-instance of": Q is a τ-instance of Q' iff there exists a substitution η such that $Q =_\tau Q'\eta$.*
- *The relation "is a τ-generalization": Q is a τ-generalization of Q' iff Q' is a τ-instance of Q.*

Example 4 (Example 3 continued). Since $\tau = \langle append \mapsto \longrightarrow 2\{\}\rangle$, we do not care of what happens to the second argument: $append([1], 2, [3, 4])$ is a τ-instance of $append([1|x], f(x), [3|z])$, with $\eta = \{x/[], z/[4]\}$. Otherwise stated, the atom $append([1|x], f(x), [3|z])$ is a τ-generalization $append([1], 2, [3, 4])$.

Finally we give a bunch of obvious definitions:

Definition 9 (Ordering sets of positions).

- $\tau \subseteq \tau'$ *if for each relation symbol* p *in* Π, $\tau(p) \subseteq \tau'(p)$.
- $\tau \subset \tau'$ *if* $\tau \subseteq \tau'$ *and* $\tau \not\models \tau'$.
- τ_{min} *is the set of positions verifying: for each* p *in* Π, $\tau_{min}(p) = \varnothing$ *and* τ_{max} *is the set of positions verifying: for each* p *in* Π, $\tau_{max}(p) = [1, arity(p)]$.

4.2 DN Sets of Positions

We give here a precise definition of the kind of arguments we are interested in. The name "derivation neutral" stems from the fact that τ-arguments do not play any rôle in the derivation process. The next subsection formalizes this intuition.

Definition 10 (Derivation Neutral). *A set of positions* τ *is* DN *for a clause* $p(s_1, \ldots, s_n) \leftarrow Body$ *if:*

$$\forall i \in \tau(p), \begin{cases} s_i \text{ is a variable} \\ s_i \text{ occurs only once in } p(s_1, \ldots, s_n) \\ \text{for each } q(t_1, \ldots, t_m) \in Body,\ [s_i \in Var(t_j) \Rightarrow j \in \tau(q)]\ . \end{cases}$$

A set of positions is DN for a logic program P *if it is DN for each clause of* P.

Example 5 (Example 4 continued). The set of positions $\tau = \langle append \mapsto \longrightarrow 2\{\}\rangle$ is DN for `C2`, the recursive clause defining *append*, but is *not* DN for the program `APPEND` since `Ys` appears twice in the unit clause `C1`.

The preceding notion is closed under renaming:

Proposition 1. *Let* τ *be a set of positions,* c *be a clause, and* c' *be a variant of* c. *If* τ *is DN for* c *then* τ *is DN for* c'.

4.3 Left Derivation and DN Sets of Positions

Our goal here is to generalize the Lifting Theorem of Logic Programming (see Sections 3.4 and 3.5 of [1], p. 56–60) in the following sense: while lifting a left derivation, we may safely ignore derivation neutral arguments which can be instantiated to any terms. As a consequence, loop detection with DN sets of positions generalizes loop detection with the subsumption test (take $\tau := \langle p \mapsto \longrightarrow \varnothing\rangle$ for any p). Proofs can be found in the long version of this paper available at `www.univ-reunion.fr/~gcc/papers`.

Theorem 2 (τ-Lifting). *Let* $\xi := Q \underset{c_1}{\Longrightarrow} Q_1 \underset{c_2}{\Longrightarrow} Q_2 \underset{c_3}{\Longrightarrow} \cdots$ *be a left derivation and* Q' *be a* τ*-generalization of* Q*. Then there exists a left derivation* $\xi' : Q' \underset{c_1}{\Longrightarrow} Q'_1 \underset{c_2}{\Longrightarrow} Q'_2 \underset{c_3}{\Longrightarrow} \cdots$ *where for each* $Q_i \in \xi$*, the corresponding* Q'_i *is a* τ*-generalization of* Q_i*.*

Example 6. Let P be a logic program. Let τ be a DN set of positions for P, with $\tau(p) = \{2\}$. Assume that there exists a successful left derivation of $\{p(s,t)\} \cup P$. Then we hold a similar left derivation when generalizing s, whatever the second argument is: for any term s' (including s) which generalizes s, for *any* term $u \in TU_{\mathcal{L}}$, there exists a left derivation of $\{p(s',u)\} \cup P$.

As a consequence of Theorem 2, we get the following result that we use in the correctness proofs of the algorithms of Section 5.

Corollary 1. *If* $(A \overset{*}{\underset{P}{\Longrightarrow}} B_1, \mathbf{B}_1 \ ; \ B_1 \overset{+}{\underset{P}{\Longrightarrow}} B_2, \mathbf{B}_2)$ *with* B_1 *a* τ*-instance of* B_2*, then* $P \cup \{A\}$ *left loops.*

Example 7. Let $c := p(f(X),Y) \leftarrow p(X,g(Y))$ be a binary clause. Then $\tau := \langle p \mapsto \{2\} \rangle$ is a DN set of positions for c. We have $p(f(X),Y) \overset{\theta}{\underset{c}{\Longrightarrow}} p(X',g(Y'))$ where $p(f(X'),Y') \leftarrow p(X',g(Y'))$ is the input clause and $\theta = \{X/X', Y/Y'\}$. Applying Corollary 1, as $p(f(X),Y)$ is a τ-instance of $p(X',g(Y'))$, we get that $p(f(X),Y)$ left loops w.r.t. $\{c\}$. We point out that we do not get this result from the classical Lifting Theorem as $p(f(X),Y)$ is not an instance of $p(X',g(Y'))$.

4.4 DN Sets of Positions for Binary Programs

We present below an algorithm for computing DN sets of positions. Its correctness is discussed in the long version of this paper. We shall show in the next section that we can incrementally build selected sets of binary clauses, together with their corresponding DN sets of positions. So, although the algorithm below can be generalized to arbitrary logic programs, we only consider binary programs, *i.e.*, finite sets of binary clauses.

`dna`$(BinProg, \tau)$:

in: $BinProg$: a finite set of binary clauses
τ: a set of positions

out: a DN set of positions $\tau' \subseteq \tau$

1: $\tau' := \tau$
2: **while** `dna_one_step`$(BinProg, \tau') \neq \tau'$ **do**
3: $\tau' :=$`dna_one_step`$(BinProg, \tau')$
4: **return** τ'

```
dna_one_step(BinProg, τ):
τ' := τ
for each p(s_1, ..., s_n) ← q(t_1, ..., t_n') ∈ BinProg do
    E := {i ∈ [1, n] | s_i is a variable that occurs
                                only once in p(s_1, ..., s_n)}
    F := ∅
    for each i ∈ τ'(p) ∩ E do
        for each j ∈ [1, n'] \ τ'(q) do
            if s_i ∈ Var(t_j) then F := F ∪ {i}
    τ'(p) := (τ'(p) ∩ E)\F
return τ'
```

Example 8 (Example 2 continued). $\texttt{dna}(\{\texttt{C2}\}, \tau_{max}) = \langle append\ \ \rightarrow 2\{\}\rangle$.

5 Inferring Looping Modes

In this section, we give a set of algorithms that allow to infer looping modes for the predicate symbols defined in a given logic program.

Let P be a logic program, parametric for the subsections which follow.

5.1 Looping Modes from One Binary Clause

Let p be an n-ary relation symbol and m_p be a mode for p. Suppose that we want to prove that m_p is looping. Assume that we hold a binary clause $c := p(s_1, \ldots, s_n) \leftarrow p(t_1, \ldots, t_n) \in bin\_unf(P)$ and τ a DN set of positions for c. Very intuitively, we have $p(s_1, \ldots, s_n) \underset{c}{\Longrightarrow} p(t_1, \ldots, t_n)$. If $p(s_1, \ldots, s_n)$ is a τ-instance of $p(t_1, \ldots, t_n)$, then, by the τ-Lifting Theorem 2, there exists a query Q_1 such that $p(t_1, \ldots, t_n) \underset{c}{\Longrightarrow} Q_1$ and $p(t_1, \ldots, t_n)$ is a τ-instance of Q_1. So, by Corollary 1, $p(t_1, \ldots, t_n)$ loops w.r.t. $\{c\}$. Now, to show that m_p is a looping mode, we can try to instantiate the variables of $\{s_i | i \in m_p\}$ by a grounding substitution σ. If σ only affects the arguments of $p(t_1, \ldots, t_n)$ that are neutral with respect to derivation, then, by the τ-Lifting Theorem 2, $p(t_1, \ldots, t_n)\sigma$ also loops w.r.t. $\{c\}$. But, very intuitively, we have $p(s_1, \ldots, s_n)\sigma \underset{c}{\Longrightarrow} p(t_1, \ldots, t_n)\sigma$. Therefore, $p(s_1, \ldots, s_n)\sigma$ loops w.r.t. $\{c\}$. Hence, as $c \in bin\_unf(P)$, $p(s_1, \ldots, s_n)\sigma$ loops w.r.t. $bin\_unf(P)$, so, by Theorem 1, $p(s_1, \ldots, s_n)\sigma$ left loops w.r.t. P. Hence m_p is a looping mode.

The function `unit_loop` below relies on the above remarks. Its termination relies on that of `dna` and its partial correctness follows from that of `dna` and the result:

Theorem 3. *Let $p \in \Pi_P$, m_p be a mode of p, and $c \in bin\_unf(P)$.*
If $\texttt{unit\_loop}(m_p, c) \not\models \textbf{false}$, *there exists $A \in [m_p]$ such that A left loops w.r.t. P.*

`unit_loop`(m_p, c):

in: m_p: a mode of p and c: a binary clause $\in$ *bin_unf*(P)
out: a pair $(\tau, \{c\})$, where τ is a DN set of positions for $\{c\}$ if c allows to classify m_p as a looping mode or **false**

1: $p(s_1, \ldots, s_n) \leftarrow q(t_1, \ldots, t_{n'}) := c$
2: $\tau :=$ `dna`$(\{c\}, \tau_{max})$
3: **if** $p(s_1, \ldots, s_n)$ is a τ-instance of $q(t_1, \ldots, t_{n'})$ **and** $Var(\{s_i \mid i \in m_p\}) \cap Var(\{t_i \mid i \not\in \tau(p)\}) = \varnothing$
4: **then return** $(\tau, \{c\})$
5: **else return false**

5.2 Looping Modes from a Set of Binary Clauses

We now introduce a data structure which we call *dictionary* and that we use both in the algorithms that follow and in their correctness proofs. A dictionary is a set of tuples $(Atom, \tau, BinProg)$ where $Atom \in TB_{\mathcal{L}}$, $BinProg$ is a set of binary clauses and τ a set of positions. Moreover:

Definition 11 (D). *A dictionary Dict enjoys the property* **D** *if Dict is a finite set such that for any* $\big(Atom, \tau, BinProg\big) \in Dict$ *we have: BinProg is a finite subset of bin_unf*(P)*,* τ *is DN for BinProg, and Atom loops w.r.t. BinProg.*

Assume we hold: a tuple $(q(u_1, \ldots, u_{n'}), \tau, BP)$ of a dictionary satisfying **D**; a binary clause $p(s_1, \ldots, s_n) \leftarrow q(t_1, \ldots, t_{n'}) \in$ *bin_unf*(P); and we would like to prove that a given mode m_p is looping. If $q(t_1, \ldots, t_{n'})$ is a τ-generalization of $q(u_1, \ldots, u_{n'})$, then, by the τ-Lifting Theorem 2, $q(t_1, \ldots, t_{n'})$ loops w.r.t. BP. Then, to show that m_p is a looping mode, we can reason as in Subsection 5.1.

The function `loop_with_dict` relies on the above remarks. Its termination relies on that of `dna` and finiteness of *Dict* and its partial correctness follows from that of `dna` and the result below.

`loop_with_dict`(m_p, c, *Dict*):

in: m_p: a mode of p, c: a binary clause $\in$ *bin_unf*(P) and *Dict*: a dictionary satisfying **D**
out: a pair $(\tau, BinProg)$, where τ is a DN set of positions for $Binprog \subseteq$ *bin_unf*(P), which allows to classify m_p as a looping mode or **false**

1: $p(s_1, \ldots, s_n) \leftarrow q(t_1, \ldots, t_{n'}) := c$
2: **for each** $\big(q(u_1, \ldots, u_{n'}), \tau, BinProg\big) \in Dict$ **do**
3: **if** $\begin{cases} q(u_1, \ldots, u_{n'}) \text{ is a } \tau\text{-instance of } q(t_1, \ldots, t_{n'}) \text{ and} \\ Var(\{s_i \mid i \in m_p\}) \cap Var(\{t_i \mid i \not\in \tau(q)\}) = \varnothing \end{cases}$
4: **then return** $($`dna`$(BinProg \cup \{c\}, \tau), BinProg \cup \{c\})$
5: **return false**

Theorem 4. *Let $p \in \Pi_P$, m_p be a mode of p, and $c \in bin\_unf(P)$. If Dict satisfies* $\boldsymbol{D}$ *and* $\texttt{loop\_with\_dict}(m_p, c, Dict) \not\models \textbf{false}$ *then there exists $A \in [m_p]$ such that A left loops w.r.t. P.*

5.3 Looping Modes for a Predicate

The function we use to infer looping modes for a predicate symbol is given in Figure 1. Our algorithm maintains the following invariant:

Lemma 1. $\boldsymbol{D}$ *always holds for $Dict'$.*

Concerning termination, note that calls to `modes`, `unit_loop`, `more_general` and `loop_with_dict` fulfill their specifications hence terminate. Since both M_p and $BinProg$ are finite sets, termination is ensured. Partial correctness is a consequence of Lemma 1 and partial correctness of the functions `unit_loop` and `loop_with_dict`.

5.4 Looping Modes for a Logic Program

The top-level function we use to infer looping modes for each predicate symbol of any logic program P is given in Figure 2. Notice that Π_P is finite and, for any

---

`infer_looping_modes_pred` ($BinProg$, p, $Dict$):

in: $BinProg$: a finite set of binary clauses $\subseteq bin\_unf(P)$,
p: a relation symbol $\in \Pi_P$ and $Dict$: a dictionary satisfying **D**

out: a pair $(L_p, Dict')$ where L_p is a looping condition for p and $Dict'$ is a dictionary satisfying **D**

1: $M_p :=$ `modes`(p), $L_p := \varnothing$ and $Dict' := Dict$
2: **for each** $c := p(s_1, \ldots, s_n) \leftarrow B \in BinProg$ **with** B / ~~true~~ **do**
3: **for each** $m_p \in M_p$ **do**
 /* NB: M_p is modified line 6 and line 11 */
4: **if** `unit_loop`(m_p, c) / ~~false~~ **then**
5: $(\tau, BP) :=$ `unit_loop`(m_p, c)
6: $M_p := M_p \backslash$`more_general`(m_p) /* cf. Definition 5 */
7: $L_p := L_p \cup \{m_p\}$
8: $Dict' := Dict' \cup \{(p(s_1, \ldots, s_n), \tau, BP)\}$
9: **elsif** `loop_with_dict`$(m_p, c, Dict')$ / ~~false~~ **then**
10: $(\tau, BP) :=$ `loop_with_dict`$(m_p, c, Dict')$
11: $M_p := M_p \backslash$`more_general`(m_p)
12: $L_p := L_p \cup \{m_p\}$
13: $Dict' := Dict' \cup \{(p(s_1, \ldots, s_n), \tau, BP)\}$
14: **return** $(L_p, Dict')$

Fig. 1. Inference of looping modes for a predicate symbol.

`infer_looping_modes_prog`(P, max):

in: P: a logic program and max: an non-negative integer
out: a set of pairs (p, L_p) where, for each $p \in \Pi_P$, L_p is a looping condition for p

1: $BinProg :=$ the binary clauses of $T_P^\beta \uparrow max$
2: $Dict := \varnothing$ and $Res := \varnothing$
3: **for each** $p \in \Pi_P$ **do**
4: $\quad (L_p, Dict) :=$ `infer_looping_modes_pred`($BinProg$, p, $Dict$)
5: $\quad Res := Res \cup \{(p, L_p)\}$
6: **return** Res

Fig. 2. The top-level function for inferring looping modes.

non-negative integer max, $T_P^\beta \uparrow max$ is a finite set $\subseteq bin\_unf(P)$. Line 2, $Dict$ is initialized to $\varnothing$ which satisfies **D**. Hence all calls to `infer_looping_modes_pred` fulfill their specification. This shows termination and partial correctness of the function `infer_looping_modes_prog`. We point out that correctness is independent of whether the relation symbols are analyzed according a topological sort of the strongly connected components of the call graph of P. However, $Dict$ is always increasing and, due to the definition of binary unfoldings, inference of looping modes is much more efficient if relation symbols are processed bottom-up.

5.5 Running the Algorithm

Example 9. We consider the program `APPEND3`:

```
append3(X,Y,Z,T):-append(X,Y,W),append(W,Z,T).
```

augmented by the `APPEND` program. $T^\beta_{\texttt{APPEND3}} \uparrow 2$ includes:

```
append([A|B],C,[A|D]):-append(B,C,D).        % C3
append3(A,B,C,D):-append(A,B,E).             % C4
append3([],A,B,C):-append(A,B,C).            % C5
```

The dictionary $Dict$, built from `C3` while processing `append`:

$$\{(append([x_1|x_2], x_3, [x_1|x_4]),\\ \tau_1 = \langle append \longrightarrow 2\{\}\rangle,\\ \{append([x_1|x_2], x_3, [x_1|x_4]) \leftarrow append(x_2, x_3, x_4)\})\}$$

shows the looping mode $\{2\}$, including, with all its τ_1-generalizations, the query: `append([A|B],void,[A|C])`. For `append3`, from `C4` and `C5`, the updated dictionary $Dict' = Dict \cup$

$$\begin{array}{l}\{(append3(x_1,x_2,x_3,x_4),\\ \qquad\qquad \tau_2 = \langle append3 \mid \longrightarrow 2\{3,4\}, append \mid \longrightarrow 2\{\}\rangle,\\ \qquad\qquad \{append3(x_1,x_2,x_3,x_4) \leftarrow append(x_1,x_2,x_5),\\ \qquad\qquad\quad append([x_1|x_2],x_3,[x_1|x_4]) \leftarrow append(x_2,x_3,x_4)\}),\\ \quad (append3([],x_1,x_2,x_3),\\ \qquad\qquad \tau_3 = \langle append3 \mid \longrightarrow 3\{\}, append \mid \longrightarrow 2\{\}\rangle,\\ \qquad\qquad \{append3([],x_1,x_2,x_3) \leftarrow append(x_1,x_2,x_3),\\ \qquad\qquad\quad append([x_1|x_2],x_3,[x_1|x_4]) \leftarrow append(x_2,x_3,x_4)\})\}\end{array}$$

allows the elimination of the looping modes $\{2,3,4\}$ and $\{1,3\}$ including the query `append3(A,void,void,void)` with all its τ_2-generalizations and the query `append3([],A,void,B)` with all its τ_3-generalizations. Note that we do not have to guess the constant [] for the last query as it appears naturally in the binary unfoldings of `APPEND3`.

6 Proving Optimality of Termination Conditions

It turns out that a slight modification of `infer_looping_modes_prog` gives a function which may prove the optimality (see Definition 6) of termination conditions (as computed by a tool for termination inference, e.g. cTI [19] or TerminWeb [13]). For each pair $(p, \varnothing)$ in the set the function returns, we can conclude that the corresponding TC_p is *the* optimal termination condition which characterizes the operational behavior of p with respect to $\mathcal{L}_{term}$. Termination and partial correctness rely on similar arguments than those in previous subsections.

`optimal_tc`(P, max, $\{TC_p\}_{p\in\Pi_P}$):

in: P: a logic program, max: an non-negative integer and $\{TC_p\}_{p\in\Pi_P}$: a set of termination conditions

out: a set of pair (p, M_p) where, for each $p \in \Pi_P$, M_p is a multi-mode of p with no information with respect to its left behavior

note: If for each $p \in \Pi_P$, $M_p = \varnothing$, then $\{TC_p\}_{p\in\Pi_P}$ is optimal

1: $BinProg := T_P^{\beta} \uparrow max$, $Dict := \varnothing$ and $Res := \varnothing$
2: **for each** $p \in \Pi_P$ **do**
3: $\quad (L_p, Dict) :=$ `infer_looping_modes_pred`($BinProg$, p, $Dict$)
4: $\quad M_p :=$ `modes`$(p)\backslash($`less_general`$(TC_p) \cup$ `more_general`$(L_p))$
5: $\quad Res := Res \cup \{(p, M_p)\}$
6: **return** Res

Example 10. We apply our algorithm to the program `APPEND3` of Subsection 5.5 (see also Example 2). We get, for `append` and `append3`:

$$\begin{aligned}
L_{append} &= \{\{2\}\} \\
\texttt{more_general}(L_{append}) &= \{\varnothing, \{2\}\} \\
TC_{append} &= \{\{1\}, \{3\}\} \\
\texttt{less_general}(TC_{append}) &= \{\{1\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}, \\
M_{append} &= \{\} \\
L_{append3} &= \{\{1,3\}, \{2,3,4\}\} \\
\texttt{more_general}(L_{append3}) &= \{\varnothing, \{1\}, \{2\}, \{3\}, \{4\}, \{1,3\}, \{2,3\}, \{2,4\}, \\
&\quad \{3,4\}, \{2,3,4\}\} \\
TC_{append3} &= \{\{1,2\}, \{1,4\}\} \\
\texttt{less_general}(TC_{append3}) &= \{\{1,2\}, \{1,4\}, \{1,2,3\}, \{1,2,4\}, \{1,3,4\}, \\
&\quad \{1,2,3,4\}\} \\
M_{append3} &= \{\}
\end{aligned}$$

Hence in both cases, we have characterized the left behavior of the predicates.

7 An Experimental Evaluation

We have implemented the algorithms presented in Sections 4, 5 and 6. Then we have applied them on some small programs from standard benchmarks of the termination analysis literature [21,2,9] (predefined predicates were erased). Here is the configuration we used for this experiment (see Table 1): PowerPC, 333MHz, 192Mb, Linux 2.2, SICStus Prolog 3.8.5, 3.8 MLips. Timings are average over 10 runs. The column *opt?* indicates whether the result of cTI is proved optimal (y) or not (*?*). The column *max* gives the least non-negative integer implying optimality (or the least non-negative integer where it seems we get the most precise information from non-termination inference). Then timings appear, followed by a pointer to a comment to the notes below.

1. The result of cTI is optimal.
2. For this program (and the following), cTI does not compute an optimal termination condition for `split`. Nevertheless, the termination condition for `mergesort` is shown optimal.
3. The result of cTI is not optimal.
4. This is an example where the binary unfoldings reveal its explosive nature. The analyzed program P (from [21], p. 64), simulates in Prolog a Turing machine. The first iteration $T_P^\beta \uparrow 1$ generates more than one hundred non-unit binary clauses. The second iteration generates more than two thousand new non-unit binary clauses.
5. `mult(s(s(0)),A,B)` and `mult(s(s(s(0))),A,void)` are detected as left looping, although the queries `mult(0,A,B)` and `mult(s(0),A,B)` do *not* left loop.

Table 1. Some De Schreye's, Apt's, and Plümer's programs.

		cTI		Optimal			
program	top-level predicate	term-cond	time[s]	opt?	max	time[s]	cf.
`permute`	permute(x,y)	x	0.24	**y**	1	0.01	
`duplicate`	duplicate(x,y)	$x \vee y$	0.09	**y**	1	0.01	
`sum`	sum(x,y,z)	$x \vee y \vee z$	0.26	**y**	1	0.01	
`merge`	merge(x,y,z)	$(x \wedge y) \vee z$	0.39	?	1	0.01	note 1
`dis-con`	dis(x)	x	0.35	**y**	1	0.01	
`reverse`	reverse(x,y,z)	x	0.14	**y**	1	0.01	
`append`	append(x,y,z)	$x \vee z$	0.14	**y**	1	0.01	
`list`	list(x)	x	0.05	**y**	1	0.01	
`fold`	fold(x,y,z)	y	0.15	?	1	0.01	note 1
`lte`	goal	1	0.19	**y**	1	0.01	
`map`	map(x,y)	$x \vee y$	0.13	**y**	2	0.01	
`member`	member(x,y)	y	0.06	**y**	1	0.01	
`mergesort`	mergesort(x,y)	x	0.75	**y**	2	0.04	note 2
`mergesort_ap`	mergesort_ap(x,y,z)	$x \vee z$	1.20	**y**	2	0.10	
`naive_rev`	naive_rev(x,y)	x	0.19	**y**	1	0.01	
`ordered`	ordered(x)	x	0.07	**y**	1	0.01	
`overlap`	overlap(x,y)	$x \wedge y$	0.08	**y**	1	0.01	
`permutation`	permutation(x,y)	x	0.25	**y**	1	0.01	
`quicksort`	quicksort(x,y)	x	0.59	**y**	1	0.03	
`select`	select(x,y,z)	$y \vee z$	0.12	**y**	1	0.01	
`subset`	subset(x,y)	$x \wedge y$	0.14	?	1	0.01	note 1
`sum_peano`	sum(x,y,z)	$y \vee z$	0.19	**y**	1	0.01	
`pl2.3.1`	p(x,y)	0	0.03	?	1	0.01	note 3
`pl3.5.6`	p(x)	x	0.09	**y**	2	0.01	
`pl4.4.6a`	perm(x,y)	x	0.19	**y**	1	0.01	
`pl4.5.2`	s(x,y)	0	0.25	**y**	1	0.04	
`pl4.5.3a`	p(x)	0	0.02	**y**	1	0.01	
`pl5.2.2`	turing(x,y,z,t)	0	3.41	?	1	0.50	note 4
`pl7.2.9`	mult(x,y,z)	$x \wedge y$	0.33	**y**	3	0.03	note 5
`pl7.6.2a`	reach(x,y,z)	0	0.22	?	1	0.01	note 1
`pl7.6.2b`	reach(x,y,z,t)	0	0.35	?	1	0.01	note 3
`pl7.6.2c`	reach(x,y,z,t)	$z \wedge t$	0.46	**y**	2	3.00	
`pl8.3.1a`	minsort(x,y)	x	0.36	**y**	2	0.04	
`pl8.4.1`	even(x)	x	0.19	**y**	1	0.01	
`pl8.4.2`	e(x,y)	x	0.78	**y**	1	0.02	

8 Conclusion

To our best knowledge, there is no other automated analysis dealing with optimality proofs of termination conditions for logic programs.

Some extensions of the Lifting Theorem with respect to infinite derivations are presented in [14], where the authors study properties of finite failure.

Loop checking in logic programming is a subject related to non-termination, where Bol [5] sets up some solid foundations (see also [23]). A loop check is a device to prune derivations when it seems appropriate. A loop checker is defined as *sound* if no solution is lost. It is *complete* if all infinite derivations are pruned. A complete loop check may also prune finite derivations. If such a mechanism is to be included into a logic programming system, then Bol advocates and studies sound loop checkers. Completeness is shown only for some restricted classes of function-free programs. Loop checking is also important for partial deduction [15]. In this case, Bol emphasizes complete loop checkers, which were also studied in [6,22]. The main difference with our work is that we want to pinpoint *some* infinite derivations that we build bottom-up. We are not interested in completeness nor in soundness. Moreover, in [11], the undecidability of the halting problem for programs with one binary clause and one atomic query is shown. This clearly puts an upper bound on what one can expect to do.

Nonetheless, we point out that the combination of termination inference and non-termination inference may give a strong result for the program being analyzed. Although the two methods are both incomplete, when their results are complementary, it implies that each analysis is optimal. Altogether they can sometimes *characterize* the operational behavior of logic programs with respect to the left most selection rule and the language used to describe classes of atomic queries.

More work is needed to refine the implementation into an efficient analyzer. In particular, the binary unfoldings need to be either computed with care or abstracted, due to the potential exponential number of binary clauses it may generate. How to take the predefined predicates into account is another problem to solve. Finally, for rational trees, we note that [8] provides an undecidable necessary and sufficient condition for the existence of a query which loops with respect to a binary clause. Moving to other constraint structures seems a worthwhile topic.

References

1. K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.
2. K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.
3. K. R. Apt and M. H. Van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.
4. T. Arts and H. Zantema. Termination of logic programs using semantic unification. In *Logic Program Synthesis and Transformation*, volume 1048 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1996.
5. R. Bol. *Loop Checking in Logic Programming.* PhD thesis, CWI, Amsterdam, 1991.
6. M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.

7. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
8. D. De Schreye, M. Bruynooghe, and K. Verschaetse. On the existence of non-terminating queries for a restricted class of Prolog-clauses. *Artificial Intelligence*, 41:237–248, 1989.
9. D. De Schreye and S. Decorte. Termination of logic programs : the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.
10. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering,Communication and Computing*, 12(1/2):117–156, 2001.
11. P. Devienne, P. Lebègue, and J-C. Routier. Halting problem of one binary Horn clause is undecidable. In *LNCS*, volume 665, pages 48–57. Springer-Verlag, 1993. Proc. of STACS'93.
12. M. Gabbrielli and R. Giacobazzi. Goal independency and call patterns in the analysis of logic programs. In *Proceedings of the ACM Symposium on applied computing*, pages 394–399. ACM Press, 1994.
13. S. Genaim and M. Codish. Inferring termination condition for logic programs using backwards analysis. In *Proceedings of Logic for Programming, Artificial intelligence and Reasoning*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2001. TerminWeb can be used online from `http://www.cs.bgu.ac.il/~codish`.
14. R. Gori and G. Levi. Finite failure is and-compositional. *Journal of Logic and Computation*, 7(6):753–776, 1997.
15. H. J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language : a theory and implementation in the case of Prolog. In *Proc. of the 9th POPL*, pages 255–267, 1982.
16. N. Lindenstrauss. TermiLog: a system for checking termination of queries to logic programs, 1997. `http://www.cs.huji.ac.il/~naomil`.
17. F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs by means of approximations. In M. J. Maher, editor, *Proc. of the 1996 Joint Intl. Conf. and Symp. on Logic Programming*, pages 7–21. MIT Press, 1996.
18. F. Mesnard and U. Neumerkel. cTI: a tool for inferring termination conditions of ISO-Prolog, april 2000. `http://www.complang.tuwien.ac.at/cti`.
19. F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. In P. Cousot, editor, *Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 93–110. Springer-Verlag, Berlin, 2001.
20. R. O'Keefe. *The Craft Of Prolog.* MIT Press, 1990.
21. L. Plümer. *Termination proofs for logic programs.* Number 446 in LNAI. Springer-Verlag, Berlin, 1990.
22. Y-D. Shen, L-Y. Yuan, and J-H. You. Loops checks for logic programs with functions. *Theoretical Computer Science*, 266(1-2):441–461, 2001.
23. D. Skordev. An abstract approach to some loop detection problems. *Fundamenta Informaticae*, 31:195–212, 1997.
24. C. Speirs, Z. Somogyi, and H. Søndergaard. Termination analysis for Mercury. In P. Van Hentenryck, editor, *Proc. of the International Static Analysis Symposium*, volume 1302 of *LNCS*, pages 160–171. Springer-Verlag, 1997.

Author Index

Adam, S. 231

Bagnara, R. 214
Ball, T. 231
Blanchet, B. 343
Bruynooghe, M. 478
Bueno, F. 102
Bultan, T. 69

Codish, M. 478
Corin, R. 327

Das, M. 231
De Schreye, D. 151
Di Pierro, A. 311

Esparza, J. 3
Etalle, S. 327

Faxén, K.F. 279
Field, J. 198

Gallardo, M.M. 396
Genaim, S. 478
Giacobazzi, R. 462
Goyal, D. 198

Hall, R.J. 2
Hankin, C. 311
Heckmann, R. 295
Hill, P.M. 214
Hymans, C. 445

Jeannet, B. 52

King, A. 85

Lam, M.S. 182
Langenbach, M. 295
Lee, C.S. 494
Lerner, S. 231
Lu, L. 85
Lumsdaine, A. 264

Manevich, R. 198
Mantel, H. 377
Martel, M. 133
Massé, D. 429
Merino, P. 396
Mesnard ,F. 510
Miné, A. 117
Müller-Olm, M. 4

Neumerkel, U. 510
Nielson, F. 20
Nielson, H.R. 20

Payet, E. 510
Pimentel, E. 396
Probst, C.W. 167

Rajamani, S.K. 231
Ramalingam, G. 198
Ranzato, F. 412, 462
Reps, T. 1
Ricci, E. 214
Ruf, E. 248

Sabelfeld, A. 377
Sagiv, M. 198
Seidl, H. 4, 20
Seigle, M. 231
Serebrenik, A. 151

Tapparo, F. 412
Thesing, S. 295

Vanhoof, W. 478
Vaucheret, C. 102
Veldhuizen, T.L. 264
Venet, A. 36

Weimer, W. 231
Whaley, J. 182
Wiklicky, H. 311

Yavuz-Kahveci, T. 69

Zaffanella, E. 214
Zanotti, M. 361